NASA Contractor Report 178386

The Computational Structural Mechanics Testbed Architecture: Volume III - The Interface

Carlos A. Felippa

Lockheed Missiles and Space Company, Inc.
Palo Alto, California

Contract NAS1-18444

December 1988

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Preface

This five-volume document presents **Command Language for Applied Mechanics Processors (CLAMP)**. As the name suggests, CLAMP is designed to control the flow of execution of Processors written for the **Network of Interactive Computational Elements (NICE)**, an integrated software system developed at the Applied Mechanics Laboratory.

The syntax of CLAMP is largely based on a 1969 command language (NOSTRA Input Language (NIL)). The language is written in the form of free-field source command records. These records may reside on ordinary text files, be stored as global database text elements, or be directly typed at a terminal. These source commands are read and processed by an interpreter **Command Language Interface Program (CLIP)**. The output of CLIP does not have meaning *per se*. The Processor that calls CLIP is responsible for translating the decoded commands into specific actions.

NIL and its original interpreter "LODREC," which now constitutes the "kernel" of CLIP, has been put to extensive field testing for over a decade. NIL has been the input language used by all application programs developed by the author since 1969 to 1979. (LODREC also drives the relational data manager RIM developed by Boeing for NASA LaRC.) During this period many features of varying degree of complexity were tried and about half of them discarded or replaced after extensive experimentation. CLAMP represents a significant enhancement of NIL, particularly as regards to directive processing, interface with database management facilities, and interprocessor control. The current version is therefore believed to be powerful, efficient, and easy to use, and well suited to interactive work.

Volume I (NASA CR 178384) presents the basic elements of the CLAMP language and is intended for all users. Volume II (NASA CR 178385), which covers CLIP directives, is intended for intermediate and advanced users. Volume III (NASA CR 178386) deals with the CLIP-Processor interface and related topics, and is meant only for Processor developers. Volume IV (NASA CR178387) describes the Global Database Manager: GAL-DBM and Volume V (NASA CR178388) describes the Input-Output Manager: DMGASP.

All volumes are primarily organized as reference documents. Except for feeble attempts here and there (*e.g.* §3.1 in Volume I and Appendix A in Volume III), the presentation style is not tutorial.

Acknowledgements

The ancestor of CLIP, LODREC, was patterned after the input languages of ATLAS and SAIL, two structural analysis codes that evolved at Boeing in the late 1960s. Newer language capabilities, notably command procedures and macrosymbols, have been strongly influenced by the Unix™ operating system and the C programming language, as popularized by Kernighan, Plauger and Ritchie in their textbooks. The Unix "shell/kernel" concept permeates the architecture of the NICE system, of which CLIP is a key component.

The author is indebted to the many CLIP users for constructive criticism and suggestions that have resulted in a steady improvement of the interpreter, the CLAMP language, and its documentation over the past four years. Special thanks are due John DeRuntz, Don Flaggs, Bill Greene, Stan Jensen, Peter Kellner, Warren Hoskins, Tina Lotts, Ian Mathews, Bill Loden, Charles Perry, Charles Rankin, Jan Schipmolder, Gary Stanley, Brian Stocks, Lyle Swenson, Phil Underwood, Frank Weiler and Jeff Wurtz. Dave Cunningham contributed VAX/VMS environment query routines.

The development of CLIP during the period 1980-1981 was supported by the Advanced Software Architecture Project of the Independent Research Program of Lockheed Missiles and Space Co., Inc. The support received from 1982 to date from MSD's Structures is gratefully acknowledged. The development of several CLIP enhancements reported here has been supported by NASA Langley Research Center on contract NAS1-17660.

Contents

THIS PAGE LEFT BLANK INTENTIONALLY.

1

Introduction

§1.1 THE CLIP-PROCESSOR INTERFACE

The running Processor communicates with CLIP through entry points provided in the CLIP shell. The entry points are implemented as FORTRAN 77 functions or subroutines. All communication data are passed through arguments or function returns. There is no communication through common blocks, which would degrade modularity. The set of entry points constitutes the **CLIP-Processor Interface**. The description of this interface is the main topic of the present Volume.

The entry points described here can be classified in three types:

CLIP Control. Calls to these entry points control subsequent command-loading actions. These entry points are alphabetically listed in Table 1.1 and described in Section 2. By far the most important is `CLREAD`, which directs CLIP to load the next command; it supersedes the old entry point `CLNEXT`. The other entry points in this class are primarily for advanced applications.

2. *Item Processing.* A command has been read in and decoded by CLIP (generally in response to a `CLREAD` request). Next, the command interpreter shell of the Processor accesses keywords, qualifiers and data values so as to carry out the actions requested by the user. A fairly large number of entry points is provided for convenient implementation of these functions. These entry points are summarized in Tables 1.2 through 1.7, which are grouped in accordance to the organization of later Sections of this Volume.

3. *Miscellaneous Utilities.* These entry points provide miscellaneous services that do not fall within the preceding two classes. For example: getting run information, evaluating macrosymbols, converting characters to Hollerith and vice versa, comparing keywords. Some of these services are not necessarily tied to CLIP, but involve more primitive actions. These entry points are summarized in Tables 1.8 through 1.10.

The material described in the following sections is primarily intended for ***processor developers***, and not for the general public. Accordingly, a fairly high level of proficiency with FORTRAN 77 is assumed.

If you are a processor developer, one important thing to keep in mind is that the Processor-CLIP relationship is of *master-slave* type. More precisely,

> *Your Processor is the Master*
> *CLIP is the Slave*

That is, your Processor (or, more precisely, the Processor Executive) can invoke CLIP, but the opposite is not true.

Of course, the control hierarchy is affected by the presence of other elements, such as user procedures and command procedures. Nevertheless, from a technical standpoint the master-slave relationship holds.

Table 1.1 **CLIP Control Entry Points**

Entry Point Name	*Purpose*	*Section in which described*
CLGET	Get next command image	§2.3
CLPUT	Insert immediate one-line message	§2.4
CLPUTM	Insert multiline message	§2.5
CLPUTW	Insert one-line message and wait	§2.6
CLREAD	Get and parse next command	§2.7
CLNEXT	Same as CLREAD	§2.7

Table 1.2 **Entry Points for Searching**

Entry Point Name	*Purpose*	*Section in which described*
ICLSEK	Search for keyword	§5.2
ICLSEQ	Search for qualifier	§5.3
ICLKYP	Search for keyword position	§5.4
ICLQLP	Search for qualifier position	§5.5

Table 1.3 **Entry Points for Loading Individual Item Values**

Entry Point Name	*Purpose*	*Section in which described*
CCLVAL	Get character value of item	§6.2
DCLVAL	Get double-precision floating value of item	§6.3
FCLVAL	Get single-precision floating value of item	§6.4
ICLVAL	Get integer value of individual item	§6.5
NCLVAL	Get nearest integer of individual item	§6.6
XCLVAL	Get single-precision complex value of item pair	§6.7
ZCLVAL	Get double-precision complex value of item pair	§6.8

Table 1.4 **Entry Points for Loading Item List Values**

Entry Point Name	*Purpose*	*Section in which described*
CLVALC	Load character list	§7.2
CLVALD	Load double-precision floating list	§7.2
CLVALF	Load single-precision floating list	§7.4
CLVALI	Load integer list	§7.2
CLVALN	Load nearest-integer list	§7.2

Table 1.5 **Entry Points for Loading Keywords & Qualifiers**

Entry Point Name	*Purpose*	*Section in which described*
CLOADK	Load keywords	§8.2
CLOADQ	Load qualifiers	§8.3
CCLKEY	Get keyword given position	§8.4
CCLQUL	Get qualifier given position	§8.5
ICLNKY	Get number of keywords	§8.6
ICLNQL	Get number of keywords	§8.7

Table 1.6 **Entry Points for Retrieving Item Information**

Entry Point Name	*Purpose*	*Section in which described*
CCLPRE	Get item prefix	§9.2
CCLSEP	Get item separator	§9.3
ICLIST	Get list length	§9.4
ICLNIT	Get number of items	§9.5
ICLOAD	Get current load pointer	§9.6
ICLTYP	Get item type code	§9.7

Table 1.7 **Entry Points for Miscellaneous Operations**

Entry Point Name	*Purpose*	*Section in which described*
CLEIHF	Get information on specific error	§10.2
CLERR1	Get error counters	§10.3
CLGLIM	Get last image loaded	§10.4
CLSLIM	Show last image loaded	§10.5
CLSLOP	Set load pointer	§10.6

Table 1.8 **Entry Points for Retrieving Run Information**

Entry Point Name	*Purpose*	*Section in which described*
CLCHAR	Get information on control characters	§11.2
ICLRUN	Get information on run state and parameters	§11.3
ICLUNT	Get information on logical unit	§11.4

Table 1.9 **Entry Points for Evaluating Macrosymbols and Expressions**

Entry Point Name	*Purpose*	*Section in which described*
CCLMAC	Evaluate character macrosymbol	§12.2
DCLMAC	Evaluate double-precision floating macrosymbol	§12.2
FCLMAC	Evaluate single-precision floating macrosymbol	§12.2
ICLMAC	Evaluate integer macrosymbol	§12.2
NCLMAC	Evaluate nearest-integer macrosymbol	§12.2

THIS PAGE LEFT BLANK INTENTIONALLY.

2

Control Entry Points

§2.1 GENERAL DESCRIPTION

The control entry points presented in this Section are used to retrieve commands and to submit messages.

For convenience we recapitulate some of the basic terminology already discussed at length in Volume I.

> A *command* can be an *ordinary command* or a *directive.*
>
> *Ordinary commands* are handled by the Processor Executive.
>
> *Directives* are handled internally by CLIP.
>
> Commands may be submitted by either the User or the Processor.
>
> A *message* is a command submitted by the Processor.

Retrieving Commands

Entry points CLGET and CLREAD are used to "get the next ordinary command". As noted in Volume I, any directive encountered along the way is processed by CLIP; control does not return to the Processor until the next ordinary command has been found in a buffer area known as *dataline collector* (see §2.2).

If you call CLREAD, the command items are processed by CLIP and stored in the Decoded Item Table described in §4.1. (CLREAD replaces the entry point CLNEXT, which nonetheless will be retained in future versions of CLIP.)

If you call CLGET, the command image will be furnished to you untouched (except for the substitution of macrosymbols and formal procedure arguments). In general CLREAD should be used unless there is a good reason to do otherwise. Two cases in which CLGET find application are:

1. You want to do your own item parsing because the rules used by CLIP conflict with the ones you prefer.
2. You are attaching CLIP to an existing command-driven program. In this case all you need to do is "disconnect" the old "read card" statement(s) and feed the images provided by CLGET instead.

Sending Messages

Three entry points are provided to send messages: CLPUT, CLPUTW and CLPUTM. These replace the old entry point CLMAIL, which should be viewed as obsolete.

If your message is a "one-liner," as the great majority are, you should use either CLPUT or CLPUTW. Your message is then placed in front of the dataline collector as described in §2.2. No input/output takes place (as it was the case under CLMAIL); just a memory-to-memory copy.

You should use CLPUT to send messages containing one or more directives to be processed *immediately* by CLIP, before it returns control back to the calling program. But if the message contains an ordinary command, you should use the "put and wait" entry point CLPUTW instead. If you call CLPUTW, CLIP will not process the message text immediately but simply stores it and returns control to the calling program; the message will be accessed by the next CLREAD or CLGET call. Further operational details are provided in §2.2.

On rare occasions, a multiline message must be sent as a block, and several one-line messages will not do the job properly. For this situation CLIP provides CLPUTM, which opens a scratch file and writes the message lines to it. When the end of the message is signalled, CLPUTM rewinds the file and "adds" it to the command source stream just like an ADD directive would.

§2.2 COMMAND DYNAMICS

The Dataline Collector

Understanding "command dynamics" for complex situations requires some knowledge of the existence of the *dataline collector* and how commands are entered and removed from it. This will be explained in this subsection using examples to illustrate the basic procedures.

The dataline collector, often referred to as "collector" for brevity, is a long character string that holds text of commands read from the command source file as well as messages sent by the Processor.

The collector functions as a staging area that buffers fluctuations in activity. For example, if an arriving data line contains several commands, all of them are placed in the collector to wait for extraction. If a command extends over several data lines, the entire text is accumulated in the collector and continuation marks erased. Commands are extracted from the *front* of the collector, one at a time, in response to calls by the Processor.

Commands that arrive from the source file are *queued* and treated on a first-in, first-out basis. One-line messages that arrive from the Processor via CLPUT or CLPUTW are *stacked* and treated on a last-in, first-out basis. Messages that arrive via CLPUTM are queued.

User Commands

To facilitate visualization, all examples given below assume conversational operation: there is a user sitting at a terminal who types commands in response to CLIP prompts.

When the Processor starts up, the collector is empty. Suppose that CLIP is first entered by a CLREAD call. CLIP prompts the user, who responds by typing *in the same line* three commands: an ordinary command, UC_1, a directive, UD_1, and another ordinary command, UC_2:

$$UC_1 \; ; \; UD_1 \; ; \; UC_2$$

(UC and UD are used to denote "ordinary user command" and "user directive", respectively.) Since CLREAD was called, CLIP removes UC_1 from the collector and parses it. On exit from CLREAD, the collector configuration is

$$UD_1 \; ; \; UC_2 \qquad (1)$$

with the parsed contents of UC_1 in the Decoded Item Table discussed in §4.1.

Upon processing command UC_1, the Processor calls CLREAD again. Since the collector is not empty, CLIP does *not* prompt the user for more data. First it processes directive UD_1, which is removed from the collector (for simplicity, let us assume that the directive does not affect subsequent command reading, as a PROCEDURE or ADD directive would). Then CLIP removes UC_2 and parses it. On return from CLREAD, the collector is empty.

When CLREAD is called a third time, CLIP notices that the collector is empty and prompts the user for more data. In response the user types two commands: UC_3 and UC_4 in the same line. CLIP extracts UC_3 and parses it. On return from CLREAD, the collector configuration is

$$UC_4$$

with UC_3's parsing in the Decoded Item Table. On the next entry, the user is not prompted for data; UC_4 is removed and parsed, and so on.

The key feature of this procedure is that *each reference to* CLREAD *retrieves one and exactly one user command*: the i^{th} call retrieves UC_i. It doesn't matter if the user types one command per line, one command over many lines, or many commands per line; what the Processor "sees" is always the sequence

$$UC_1 \ ; \ UC_2 \ ; \ UC_3 \ ; \ UC_4 \ \ldots$$

Furthermore, the presence of directives (more precisely, directives that do not modify the command stream) does not affect what the Processor receives.

REMARK 2.1

Replacing CLREAD by CLGET in the above narrative does not change things in any essential way; it only influences the "packaging" of the information received by the Processor.

REMARK 2.2

The sequence of events is also identical if the human user is replaced by a script file or a command procedure. Of course, in such a case CLIP does not issue prompts.

Immediate Messages

The presence of messages sent by the Processor may introduce complications. Consider first one-line messages, which are the ones most frequently used.

Let's go back to the collector configuration(1)(page 2-4) that follows the first CLREAD. Before calling CLREAD again, suppose that the Processor sends through CLPUT a message containing two directives: PD_1 and PD_2 (where PD stands for "processor directive"). The message is stacked in front of the collector, so at that point the collector configuration becomes

$$PD_1 \ ; \ PD_2 \ ; \ UD_1 \ ; \ UC_2$$

Since CLPUT has been called, CLIP proceeds to digest the message immediately. It extracts PD_1 and PD_2 (in that order) from the collector, and performs the tasks indicated in them. On return from CLPUT, the collector configuration is again given by collector configuration (1). The next CLREAD call then processes UD_1 and UC_2, so there is no change with respect to the no-message case.

REMARK 2.3

Certain directives, notably **ADD** and **CALL**, erase the collector. If they arrive as messages, commands already there are lost.

Deferred Messages

But now assume that the message contains an ordinary command, PC_1, which is submitted via the "put and wait" entry point **CLPUTW**. As before, the message is stored in front of the collector:

$$PC_1 \; ; \; UD_1 \; ; \; UC_2$$

but now CLIP does not process the message; it returns leaving the collector in the preceding state. Now the next **CLREAD** call retrieves PC_1 and *not* UC_2. If no other messages intervene, the second user command is retrieved by the *third* **CLREAD** call. The user is of course unaware of this reshuffling, but the Processor logic must account for these variations as appropriate.

REMARK 2.4

You may want to work out what happens if PC_1 were sent through **CLPUT** instead of **CLPUTW**. After the next **CLREAD** call, you will find that PC_1 has disappeared without a trace! This is the reason for recommending the use of put-and-wait. There are a few cases, however, in which the submitting an ordinary command as immediate message has applications in conjunction with **CLGET**.

REMARK 2.5

If you call **CLPUTW** twice in a row to submit ordinary commands, the order is reversed. Let's say the first call sends PC_1 and the second one PC_2. As these commands are ***stacked***, the last one is processed first; *i.e.*, in the order PC_2, PC_1. The same "reversal" occurs for more calls. Should this be undesirable, either reverse the order of the calls, or send them as components of a single message.

Multiline Messages

Multiline messages are less frequently used than one-line messages so there is no need to delve into much detail here. The key difference is that the contents of a message submitted through **CLPUTM** are processed as if they appeared *on the back of the collector.*

If the collector is empty when the message is sent, there is no difference between **CLPUTW** and **CLPUTM**, because multiline messages are always made to wait. But if the collector is nonempty the event sequence may be quite different.

For example, let us say that directives PD_1 and PD_2 are submitted through **CLPUTM** when the collector has the configuration(1)(page 2-4). Upon return the state may be visualized as

$$UD_1 \; ; \; UC_2 \; ; \; PD_1 \; ; \; PD_2$$

in which PD_1 and PD_2 are not physically in the collector but in a card-image file to be added once UC_2 is removed. The next (second) **CLREAD** call will remove UD_1 and UC_2, so directives PD_1 and PD_2 will not be processed until the *third* **CLREAD** call. (You should contrast this to the one-line message submission, in which PD_1 and PD_2 are processed ahead of UD_1 and UC_2.)

If the user had entered UC_1 and UC_2 on different lines, the sequence of events would be different. Because of these unpredictable side effects, it is best to avoid multiline messages if possible, or to precede them with an **EOL** directive, which empties the dataline collector (see Volume II).

§2.3 GET NEXT COMMAND IMAGE: CLGET

Entry point CLGET should be called instead of CLREAD if you want to retrieve command images and inhibit CLIP parsing. You provide a receiving character argument into which CLGET stores the image of the last command loaded. Two reasons for this ***modus operandi*** are discussed in §2.1.

Calling sequence

```
CALL CLGET (PROMPT, SPLASH, IMAGE, NCH)
```

Input arguments

PROMPT — Same as for CLREAD; cf. §2.7.

SPLASH — Same as for CLREAD; cf. §2.7.

Output arguments

IMAGE — A character string into which CLGET places the image of the next ordinary command found in the collector (cf. §4.2). The number of characters transferred into IMAGE cannot exceed the passed length; consequently, if the length is insufficient the image may be truncated.

NCH — The absolute value of NCH is the number of characters loaded into IMAGE. A negative value indicates that the command image has been truncated because its length exceeds that of the passed length of IMAGE.

REMARK 2.6

Macrosymbols in the command input are normally replaced by their values in the returned image. This default mode may be altered by sending a SET MODE directive.

REMARK 2.7

If lines are being read from a command procedure, formal argument references are normally replaced by their values in the IMAGE text. This default mode may be altered by sending a SET MODE directive.

REMARK 2.8

Directives inserted before the next user command are processed by CLIP and *not* returned in IMAGE. Thus the operation of CLGET is externally the same as that of CLREAD. (The user cannot tell the difference between the two).

EXAMPLE 2.1

Consider the following code block:

```
CHARACTER*80  IMAGE
...
CALL  CLGET (' Enter Command>', ' ', IMAGE, N)
PRINT *, IMAGE(1:N)
```

In response to the prompt, the user types

```
Enter Command> DO WHAT FOLLOWS
```

(The blank after > is part of the prompt, not of the response.) On return from **CLGET**, argument **IMAGE** will contain **DO WHAT FOLLOWS** and argument **N** will be set to 15, which is the length of the command line.

Now suppose that **IMAGE** had been declared **CHARACTER*12**, or that the third argument in the call to **CLGET** had been **IMAGE(1:12)**. Then the returned command is truncated to 12 characters:

```
DO WHAT FOLLOWS
```

and N returns -12.

§2.4 SEND ONE-LINE IMMEDIATE MESSAGE: CLPUT

The CLPUT entry point is used to send a one-line message containing directives to be interpreted by CLIP. The message may contain more than one directive, but should not normally contain user commands.

Calling sequence

```
CALL  CLPUT (TEXT)
```

Input argument

TEXT	The text of the message. It should not exceed 480 characters. The text may contain one directive, or several directives separated by semicolons.

REMARK 2.9

If you want to send ordinary commands through this entry point, be sure you understand the material presented in §2.2.

EXAMPLE 2.2

Consider the call

```
CALL  CLPUT (' *set echo = on,verbose ; *show css ')
```

This call sends a line that contains two directives to be immediately processed by CLIP.

§2.5 SEND MULTILINE MESSAGE: CLPUTM

There are occasions in which the message to be sent is so voluminous that it must be broken down into several lines of text. In this case the proper message entry point to call is CLPUTM. This is normally done in a loop, one line per call, and the last line is signalled by an end-of-message flag.

Calling sequence

```
CALL  CLPUTM (TEXT)
```

Input argument

TEXT — TEXT is a character string containing either a message line (not to exceed 80 characters) or the *EOM or *EOM/I message terminator characters. Use of the *EOM/I will cause CLIP to insert the message immediately into the command stream. Otherwise, it will simply return and a subsequent call to CLREAD or CLGET will cause the message file to be added to the command source stream.

REMARK 2.10

The end-of-message mark must be in columns 1–4 or else it will be ignored.

REMARK 2.11

If the end-of-message mark is *EOM/I, the dataline collector is flushed.

EXAMPLE 2.3

Consider

```
CALL  CLPUTM ('*PROCEDURE CHANGE.STEP (H=0.04)')
CALL  CLPUTM (' SET INTEGRATION STEP TO [H] ')
CALL  CLPUTM ('*END ')
CALL  CLPUTM ('*EOM/I ')
```

These four calls submit a three-line message that define a command procedure called **CHANGE.STEP**.

§2.6 SEND ONE-LINE MESSAGE AND WAIT: CLPUTW

This entry point operates like CLPUT, except that the message is not processed immediately by CLIP, but by the next CLREAD or CLGET call. This is the recommended procedure if the message includes one or more user commands, as explained in §2.2.

Calling sequence

```
CALL  CLPUTW (TEXT)
```

Input argument

TEXT	A character string that contains the text of the message. Its length should not exceed 480 characters.

EXAMPLE 2.4

Consider

```
CALL  CLPUTW (' set omega=1.45 ; *show macros ; set damping=.07 ')
```

On return from CLPUTW the indicated text is stored in the dataline collector, but has not been processed. Suppose that two CLREAD calls follow. The first CLREAD call "captures" the SET OMEGA command. The second call processes the SHOW directive and "captures" the SET DAMPING command.

If this call had been submitted via CLPUT followed by CLREADs, the first ordinary command would have been lost.

§2.7 READ NEXT USER COMMAND: CLREAD

You request that the next user command be read and decoded by CLIP by calling the entry point CLREAD. This is by far the most important control entry point and the only one most Processors should use.

Calling sequence

```
CALL CLREAD (PROMPT, SPLASH)
```

Input arguments

PROMPT — Optional prompt text. A character string that may contain up to 132 characters. The second through last characters will be printed as a prompting message if (a) running in conversational interactive mode and (b) CLIP expects the next command from the terminal. The extent of the prompt message is determined by its passed length, except that multiple trailing blanks, if any, are reduced to one.

The first character of PROMPT controls the spacing before the prompt line as follows:

1 One blank line.

2 Two blank lines.

\- Page skip.

Any other: no skip. This character is not printed.

A double ampersand (&&) may be used to generate a carriage return/line feed (= new line) in long prompts. This symbol is not printed. The character that follows && is not considered a space control.

This argument is ignored in the cases noted in Remark 2.12 below.

SPLASH — Optional "splash line" to be displayed before the prompt if the "verbose" mode has been enabled through a SET ECHO directive. On the VAX, this string may contain any number of characters; on other computers it is limited to 480 characters. The length of the SPLASH line is that of the passed length or that of the last nonblank character, whichever is smaller.

The first character of SPLASH is a line spacer that functions as described above for PROMPT. Double ampersands may be used to force line feeds.

This argument is ignored in the cases noted in Remark 2.13 below.

REMARK 2.12

The **PROMPT** argument is ignored in the following cases: (a) batch mode; (b) command source is not the user's terminal, or (c) the length of the **PROMPT** string is only one character.

REMARK 2.13

The **SPLASH** argument is ignored in the following cases: (a) "verbose" mode is off, or (b) any of the conditions listed in the above Remark applies.

REMARK 2.14

Entry point **CLNEXT** is a variant of **CLREAD**. It has the calling sequence

```
CALL CLNEXT (PROMPT, SPLASH, ITEMS)
```

where **ITEMS** is an output integer argument that receives the number of items in the last parsed command. This entry point will be retained in future CLIP versions.

EXAMPLE 2.5

Consider the call

```
CALL  CLREAD (' Next command:  ', ' ')
```

This call specifies

```
Next command:
```

as the prompt line (there will be a blank after the colon). The splash line is empty.

EXAMPLE 2.6

Consider now the call

```
 CALL  CLREAD (' Next command:  ',
$              ' Commands:  BEGIN, RUN, CONTINUE, STOP')
```

This call specifies the same prompt as in the previous example, but now it includes a splashline that has a short command menu.

3

Standard Command Format

§3.1 INTRODUCTION

The present Section repeats much of the material presented in §3.1 of Volume I in greater detail. As a Processor developer, you are expected to understand the precise meaning of terms such as keywords, qualifiers and item lists in order to be able to effectively use the command-processing entry points described in Sections 4 to 8 of this Volume.

The description that follows covers the so-called *standard CLAMP format.* It was noted in §3.2 of Volume I that this is a subset of the total number of command formats that CLIP can process. But many of the entry points described in following Sections are tuned to this format.

REMARK 3.1

If you decide for a command format other than the standard CLAMP format, you must be prepared to take one of the following approaches, depending on the degree of deviation from the standard.

1. *CLIP Item Parsing is Acceptable.* If the way CLIP breaks up the command into items is acceptable to you, commands may be retrieved through **CLREAD**. Command processing may be done on a detailed, item-by-item basis, avoiding search and list-loading entry points.
2. *CLIP Item Parsing is not Acceptable.* In this case you must retrieve command images through **CLGET** and do your own parsing. You may be able to send "chunks" such as item lists back to CLIP via **CLPUT** for convenient decoding. But in general this approach will entail a lot more programming work on your part, so it should be justified only under special circumstances.

If you have taken a nonstandard approach, the material that follows is not particularly relevant and may be skipped.

§3.2 PHRASES

In the standard CLAMP format, a command is always *a sequence of phrases* as shown in the display box:

$$\boxed{\textit{Standard command} \equiv \textit{Phrase}_1\ \textit{Phrase}_2\ \ldots\ \textit{Phrase}_k}$$

A Phrase (capitalized) is one or more interrelated items that must appear in sequence. A Phrase can take five forms:

Keyword
List
Keyword = *List*
Qualifier
Qualifier = *List*

(There are two variants of the second and third forms, as explained in §3.5.) The following subsections describe keywords, qualifiers and lists.

§3.3 KEYWORDS

Definition

A *keyword* is a character string that meets the following conditions:

1. It is not a component of a list.
2. It is not a qualifier.

What's the way to tell a keyword? You have to follow a process of elimination.

In the first place, the above definition says that a keyword must be a character string. A numeric item cannot be a keyword. (One may type apostrophe character strings that look like numbers, for example '1984' but this is admittedly rare.)

Next, if the alleged keyword is not the first item, look *before* it for prefixes or connectives. If you see a qualifier prefix (normally a slash) this is a qualifier and not a keyword. If you see an equals sign, this is the first item of a list (perhaps the only item). If you see a comma, this is a component of a list.

Finally, look *after* it. If you see a comma, this is a component of a list and not a keyword.

Function

Keywords are associated with *control* functions. They are used to specify operations to be performed by the Processor, and to select cases within complex operations.

Keywords should be distinguished from character string data. For example, in the two-phrase command

```
OPEN FILE = INPUT.DAT
```

`OPEN` and `FILE` are keywords, but `INPUT.DAT` is a file name and not a keyword. An easy way to distinguish the control *versus* data functions is to ask oneself: has the name **INPUT.DAT** a special significance to the Processor?

There is an ambiguous interpretation case discussed in §3.5.

The Action Verb

In the standard CLAMP format, if the first command item is a character string, it is *usually* a keyword called the *action verb*. The action verb defines what the command is supposed to do; §3.1 of Volume I contains many examples.

Note that this is recommended practice and not a mandatory rule. There is an important class of commands, called *data commands*, which consist of a list only.

§3.4 QUALIFIERS

Definition

Qualifiers are character strings (with the exception noted in the Remark below) preceded by a qualifier prefix. The default prefix is the slash, although this may be changed through a SET CHAR directive as explained in Volume II. We shall assume the slash for all examples, as in

```
OPEN /NEW
```

Here NEW is a qualifier. Note that the prefix is *not* considered part of the qualifier string. When the Processor retrieves this qualifier from CLIP, it gets NEW and not /NEW. (Doing the latter would lead to character-matching programming problems should be the prefix be changed by the user to, for example, a dollar sign.)

REMARK 3.2

Apostrophe strings should not be specified as qualifiers. For example:

```
PRINT TABLE = RRR /'Format'=E
```

should be avoided; say /FORMAT=E instead. The item-parsing logic of CLIP gets quite confused when a construction of this type is encountered.

Function

As explained in §3.1 of Volume I, the basic function of the qualifier is to implement *options*. It follows that qualifiers may always be omitted from the command, and that a default interpretation must exist.

Qualifiers may be followed by a *qualifier list* that contains one or more items. The list must be preceded by an equals sign that "attaches" it to the qualifier; this sign may not be omitted.

§3.5 ITEM LISTS

An Expanded Definition

Section 9 of Volume I defined item lists as sequence of items of the same type (numeric and character strings) separated by commas. Now the term "item sequence" denotes two or more items, as opposed to an individual item. But, for command processing tasks, it is usually convenient to view an individual item as a one-item list.

REMARK 3.3

From this expanded interpretation it follows that a *numeric item is either a list or a component of a list.*

In forming a Phrase, an item list may follow a keyword, follow a qualifier, or stand by itself. If a list follows a qualifier, it must be preceded by an equals sign. In the other cases, the equals sign is optional but never hurts. In fact, it helps to eliminate the ambiguous interpretation discussed below. These Phrase composition rules deserve a display that expands upon that of §3.2:

List
= *List*
Keyword = *List*
Keyword *List*
Qualifier = *List*

Function

Lists supply data to the Processor. Although the Processor actions may be influenced by the values of the data, the influence is not so direct and unequivocal as in the case of keywords and qualifiers.

An Ambiguous Case

There is an ambiguous case in which a character string may be interpreted as a keyword or as a one-item list. Example:

```
OPEN INPUT.DAT
```

It is clear that `OPEN`, which is the action verb, is a keyword. But what is `INPUT.DAT`? From the likely interpretation of the command "open filename", it has to be a data value; therefore it is a one-item list. But it could also be interpreted legally as a keyword according to the rules of §3.3.

Placing an equals sign between the two items would make the interpretation unambiguous:

```
OPEN = INPUT.DAT
```

But the rules of §3.4 state that the sign is optional, so the "equals-less" form cannot be ruled out.

In practice this ambiguity has not lead to significant problems in command interpretation. Therefore, the standard CLAMP format gives Processor developers a choice.

If you hate syntactical ambiguities, you may want to require users to put equals signs before character lists (or all lists, for that matter). But if you are not a particularly fastidious person and are aware that users hate typing equal signs (they are hard to find on keyboards) you may decide to accept the ambiguity in *simple* commands such as the above example. (If the ambiguity can occur in a complex command, you should redesign it.)

THIS PAGE LEFT BLANK INTENTIONALLY.

4

Item Loading Overview

§4.1 THE DECODED ITEM TABLE

When CLIP interprets an ordinary command, it stores the result of the interpretation in a *Decoded Item Table*, also called Parsed Item Table. This table was mentioned in in Remark 6.2 of Volume I to help explain some advanced concepts, but it is of limited interest to users. In the present Volume, the table is central to the exposition.

The best way to describe the configuration of the Decoded Item Table is to go through an example. Consider the eight-item command

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE
```

Upon interpreting this command, the Decoded Item Table contains the following data:

Index	*Type*	*Prefix*	*Value*	*Separator*
1	Character		SOLVE	
2	Character		FOR	
3	Character		X	=
4	Floating		0.33333333	
5	Character	/	RANGE	=
6	Integer		25	,
7	Integer		86	
8	Character	/	SCALE	

The *item index* is not stored, but serves to identify the position in the table. Attributes *type* and *value* are self-explanatory. The table also "remembers" two characters called *prefix* and *separator*, which are not part of the value itself.

Only certain special characters may legally fill the role of prefixes and separators; details to this respect have been given in Section 6 of Volume I. If none of these special characters appears, a blank value is stored.

REMARK 4.1

The Decoded Item Table may be displayed through the **SHOW DEC** directive. The display format is not exactly that shown above (for example, the value appears last to facilitate showing long character strings), but it contains the same information.

The Decoded Item Table is accessible to the Processor through the entry points described in Sections 5 to 8. The most important function of these entry points is *item loading*, which is the transfer of information from the table into the Processor work area. For example, the Processor may begin by testing the action verb, and so the keyword **SOLVE** must be retrieved. The contents of the table cannot be modified by the Processor.

§4.2 THE LOAD POINTER

Many of the Processor-CLIP interface points are of the "item loading" type. Their chief function is to facilitate transfer of information from the Decoded Item Table into the Processor work area for subsequent interpretation by the Processor executive. An item is said to have been *loaded* when it has been accessed by an item-loading function or subroutine and its value has been copied to the specified destination.

To systematize the transfer process, CLIP maintains an internal variable called the *load pointer*, which is often denoted by the FORTRAN-like symbol, `ILOAD`. This is an integer that has the index of the last item loaded, the index of a keyword or qualifier matched by a "search" operation, or the value communicated through a "set pointer" operation; whichever occurred last.

There is a closely related pointer called the *next item to load*, which is often denoted by `INEXT`. Its value is obtained by adding `1` to `ILOAD`.

The number of items in the Decoded Item Table is denoted by `ITEMS`; in the example of §4.1, `ITEMS = 8`. The load pointer `ILOAD` may range from `0` to `ITEMS`, and the next-item-to-load pointer `INEXT` from 1 to `ITEMS+1`.

The basic item-loading principle is: *select by pointing*, then *move*. More precisely, if you want to load the item that follows keyword `X`, you have to make `ILOAD` point to `X`, which in turn makes `INEXT` point to whatever follows `X`. This manipulation can be done in several ways as is summarized in §4.3.

§4.3 MANIPULATING THE LOAD POINTER

On return from a "get next command" reference to CLREAD or CLGET, ILOAD and INEXT initially have the values zero and one, respectively. From then on, the load pointer moves in response to searching, item loading and setting actions, as described next.

Searching

The occurrence of specific keywords or qualifiers in the Decoded Item Table may be tested through search operations requested via functions ICLSEK or ICLSEQ, which are described in Section 5. ICLSEK is used for keywords and ICLSEQ for qualifiers. If a match takes place, the load pointer is set to the index of the matched item. To illustrate, consider again the sample command:

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE
```

If a search is made for keyword X, the load pointer is set to 3 and INEXT becomes 4 so it points to (1/3). This happens regardless of previous operations. If no match occurs, the values do not change.

Loading an Individual Item

The value of an individual item (an isolated item or a list component) may be retrieved through the FORTRAN-callable functions named *x*CLVAL, which are described in Section 6. The single function argument is the item index with a value of zero defaulting to ILOAD. If the index is in range, upon return ILOAD points to the retrieved item and INEXT to the next one.

Loading Lists

Item lists may be processed all at once by calling subroutines named CLVAL*x*, which are described in Section 7. Loading always begins at the current INEXT and proceeds until a termination condition is reached. On return, ILOAD points to the last item transferred (if any).

Direct Setting

The load pointer may be set to a specific value by calling entry point CLSLOP, which is described in §10.3.

REMARK 4.2

The similarity of item loading and processing of direct-access files may be helpful to developers familiar with the latter. A zero ILOAD corresponds to a "file rewound" condition, a keyword search corresponds to a seek-key operation, and so on.

5
Searching

§5.1 SEARCHING: ICLSEK and ICLSEQ

This Section describes two entry points for searching the Decoded Item Table: ICLSEK for keywords and ICLSEQ for qualifiers. Both functions take two arguments. The first argument specifies the starting point for the search while the second one specifies the character string to be matched. If a match occurs the function returns the index of the matched item while ILOAD is internally set to point to it.

But what is the meaning of "to match"? The case of character-by-character equality is of course obvious: ICLSEK(1,'COPY') matches keyword COPY. But there is more to the subject than this.

Upstairs/Downstairs

First we examine the question of uppercase *vs.* lowercase. Suppose the command you have typed has the keyword

```
Copy
```

Does ICLSEK(1,'COPY') match this keyword?

Yes. Remember that CLIP converts all lowercase input to uppercase except for apostrophe strings, and that apostrophe strings should *never* be used for keywords or qualifiers; only for character data such as plot legends and the like.

Abbreviating Keywords

Many Processors do not insist on complete matching of the command keywords and qualifiers shown in the Processor Manuals or its help file. Instead, they permit ***abbreviations.*** For example, let us suppose that the Processor Manual describes a command as

```
PRINT ELEMENTS
```

but that keyword PRINT may be abbreviated to PRI (but *not* P or PR) while keyword ELEMENTS may be abbreviated to E. So in fact a user can in fact type only:

```
PRI E
```

Roots and Extensions

Abbreviations such as the ones shown above are technically known as ***keyword roots.*** The additional characters shown in the command description are called the ***extension.*** The extension is shown primarily for mnemonic purposes: PRINT ELEMENTS is more easily remembered and understood than PRI E.

How are roots and extensions specified when invoking the search functions? You should separate the two parts by a caret mark; for example:

```
ICLSEK(1,'PRI^NT')
ICLSEK(1,'E^LEMENTS')
```

tells CLIP to search for keywords PRINT and ELEMENTS, whose roots are PRI and E, respectively. Note that this convention precludes the use of the caret in keywords.

Another way of doing this is to to switch to *lowercase* for the extension, as in

```
ICLSEK(1,'PRInt')
ICLSEK(1,'Elements')
```

This notation is more readable, but has two drawbacks, one minor, one major:

1. It cannot be used on the (admittedly few) computers that do not recognize lowercase letters.
2. It may lead to ambiguity if the keyword contains nonletters. For example, if you say

```
ICLSEK (0, 'X-value')
```

   it is impossible to tell whether the dash belongs to the root or not.

Uniqueness and Consistency

A basic requirement is that keyword roots should be *unambiguous.* For example, let us say that there are two commands whose action verbs are PRINT and PROCEED. Then the abbreviations PRI and PRO are acceptable, but PR is ambiguous. The Processor developer is responsible for specifying unambiguous roots. Note that as more commands are added during the lifetime of a Processor, certain roots may have to be expanded.

A more subtle question arises when the user types "beyond" the root. For example, PRINT ELE or PRINT ELEMS.

One course of action would be to accept a keyword as long as it matches the root. This can be achieved by simply omitting the extension when you call the search functions, as in

```
ICLSEK(0, 'PRI')
ICLSEK(2, 'E')
```

This is straightforward, but may surprise and confuse some users. A more rational way is implemented in the search functions: accept a match if the excess characters agree with the extension and to disallow a match if they do not. As an illustration, any of

```
EL      ELEM      ELEMENT
```

match ELEMENTS, but ELEVATE or ELEMENTAL do not.

In practice Processor developers often follow a middle course: the tests against the extension are limited to two or three characters beyond the root. For example, if the developer writes the search reference as

```
ICLSEK (1, 'ELEM')
```

then anything that the user types beyond ELEM is ignored.

Both ICLSEK and ICLSEQ are trained to apply this approach if an extension appears in the argument. For tests-within-the-processor, the universal string-matching routine CMATCH is available.

Newer Entry Points

In addition to ICLSEK and ICLSEQ, two more search functions have been added in the present version of this document: ICLKYP and ICLQLP. These also search for given keywords or qualifiers, but return their position count rather than item index. These entry points are designed to work in conjunction with CCLKEY, CCLQUL, ICLNKY and ICLNQL, which are described in §8.

§5.2 SEARCH FOR KEYWORD: ICLSEK

Entry point ICLSEK, which is referenced as a integer function, scans the Decoded Item Table for the first occurrence of a keyword that matches its argument. The search begins at the item index specified as argument; but if this argument is zero, the "next item to load" position is assumed. If a match occurs, the function returns a nonzero value and the load pointer is set to the matched item index. If no match occurs the function returns zero and the load pointer is unchanged.

ICLSEK does not test its argument against qualifiers, items preceded by a comma or equals sign, items followed by a comma, or numeric items. None of these fits the definition of "ordinary keyword" given in §3.3.

Calling Sequence

```
M =  ICLSEK (I, KEY)
```

REMARK 5.1

An explicit function reference is rare, however; more often than not **ICLSEK** is tested within an **IF** statement.

Input Arguments

I	If I > 0, begin search at the i^{th} item. If zero, begin at INEXT.
KEY	A character string that contains the keyword to be matched left justified. The string may also specify the keyword root followed by its extension as explained in §5.1.

Function Return

ICLSEK	If a match occurs, ICLSEK returns the index of the matched item and internally sets ILOAD to point to it. If no match is detected, the function returns zero and **ILOAD** is not altered.

Procedure

Initialize function value to zero. Examine entries in the Decoded Item Table starting at I>0, or INEXT if I=0. Numeric items, qualifiers (items preceded by a qualifier prefix) and items preceded by an equals sign are skipped. A compare test of the argument and the candidate keyword is performed. If match occurs, set ILOAD to its index and return; else continue until the end of command is reached.

EXAMPLE 5.1

Assume that the last loaded command is

```
LOAD COORDINATES X = 1.2 Y = (2/3) Z = -7.5
```

and that INEXT is one. Then

```
ICLSEK (0, 'X')                 returns  3
ICLSEK (0, 'Z^-VALUE')          returns  7
ICLSEK (0, 'COOR^D')            returns  2
ICLSEK (1, 'LOAD')              returns  1
ICLSEK (2, 'LOAD')              returns  0
ICLSEK (0, 'LOADER')            returns  0
ICLSEK (0, 'LO^CK')             returns  0
ICLSEK (0, 'COOR^DINATE')       returns  2
```

EXAMPLE 5.2

This is a trickier example:

```
FILE /FILE=FILE FILE = FILE
```

with INEXT = 2. What does ICLSEK(0,'FI^LE') return? Answer: 4 (why?).

§5.3 SEARCH FOR QUALIFIER: ICLSEQ

Entry point ICLSEQ, which is referenced as a integer function, scans the Decoded Item Table for the first occurrence of a qualifier that matches its argument. The search begins at the item index specified as argument; but if this argument is zero, the "next item to load" position is assumed. If a match occurs, the function returns "next item to load" position. If a match is made, the function returns a nonzero value and the load pointer is set to the matched item index. If no match occurs the function returns zero and the load pointer is unchanged.

Calling Sequence

```
M =  ICLSEQ (I, KEY)
```

REMARK 5.2

As with ICLSEK, explicit function references are rare. More often than not ICLSEQ is tested within an IF statement.

Input Arguments

I	If I > 0, begin search at the i^{th} item. If zero, begin at INEXT.
KEY	A character array that contains the keyword to be matched left justified. It may also specify the keyword root followed by its extension as explained in §5.1.

Function Return

ICLSEQ	If a match occurs ICLSEQ returns the index of the matched item, and also internally sets ILOAD to it. If no match is detected the function returns zero and ILOAD is not altered.

Procedure

Initialize function return to zero. Examine the Decoded Item Table starting at I>0, or INEXT if I=0. Numeric items and items not preceded by a qualifier prefix are skipped. A compare test of the argument and the candidate keyword is performed. If match occurs, set ILOAD to its index and return; else continue until the end of command is reached.

EXAMPLE 5.3

Assume that the last loaded command is

```
OPEN /ROLD 3, [REAGAN]BUDGET.DEF /LIMIT=INFINITE
```

and that **INEXT** is 1. Then

```
ICLSEQ (0, 'R')          returns  2
ICLSEQ (0, 'LIM')        returns  5
ICLSEQ (0, 'L^IMIT')     returns  5
ICLSEQ (0, 'LIMITS')     returns  0
ICLSEQ (0, 'BUDGET')     returns  0
```

§5.4 SEARCH FOR KEYWORD POSITION: ICLKYP

Integer function ICLKYP receives a keyword value as argument, and returns the keyword position if that keyword occurs in the command.

Calling Sequence

```
IK =   ICLKYP (KEY)
```

Input Arguments

KEY — A character string that contains the keyword to be matched left justified. The string may also specify the keyword root followed by its extension as explained in §5.1.

Function Return

ICLKYP — Keyword position (not to be confused with the item index) if the keyword is found, otherwise it is zero. If a match occurs, pointer ILOAD is set to the keyword index.

Procedure

The Decoded Item Table is scanned from beginning to end. For each item classified as a keyword, a comparison test is performed against the argument value. If the match succeeds, the keyword position is returned. If no match is detected after scanning the entire table, a zero is returned.

EXAMPLE 5.4

Assume that the last loaded command is

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE STORE
```

Then ICLKYP('FOR') returns 2, ICLKYP('ST^ORE') returns 4, but ICLKYP('ZZZZ') returns zero.

§5.5 SEARCH FOR QUALIFIER POSITION: ICLQLP

Integer function ICLQLP receives a qualifier value as argument, and returns the qualifier position if the qualifier occurs in the command.

Calling Sequence

```
IK =   ICLQLP (KEY)
```

Input Arguments

KEY — A character string that contains the qualifier to be matched left justified. The string may also specify the qualifier root followed by its extension as explained in §5.1.

Function Return

ICLQLP — The qualifier position (not to be confused with the item index) if the qualifier is found, otherwise it is zero. If a match occurs, the pointer ILOAD is set to the item index.

Procedure

The Decoded Item Table is scanned from beginning to end. For each item classified as a qualifier, a comparison test is performed against the argument value. If the match succeeds, the qualifier position is returned. If no match is detected after scanning the entire table, a zero is returned.

EXAMPLE 5.5

Assume that the last loaded command is

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE STORE
```

Then `ICLQLP('RANGE')` returns 1, `ICLQLP('SC^ALE')` returns 2, but `ICLQLP('VOID')` returns zero.

6

Loading Individual Items

§6.1 GENERAL DESCRIPTION

The value of individual items identified by an item index may be retrieved through function entry points named *x*CLVAL. The first letter of the function name identifies the data type of the *receiving* variable in the calling program. Presently that letter may be C, F, D, I, N, X and Z, for character, single float, double float, integer, next integer, single complex and double complex data types, respectively.

The only function argument is the item index; if a value of zero is specified, INEXT is assumed.

REMARK 6.1

Some of these functions are among the most venerable pieces of code in CLIP and its ancestor LODREC. In fact, the first ICLVAL and FCLVAL were coded in 1969 on the CDC 6600, under the names IVALUE and and FVALUE, respectively. Of course there was no character data type in FORTRAN at that time; to get keywords there was an integer function HVALUE that returned a Hollerith value.

§6.2 GET INDIVIDUAL CHARACTER VALUE: CCLVAL

Entry point CCLVAL, which is invoked as a character function, returns the character value of a decoded CLIP item identified by its index.

Calling Sequence

```
CHARACTER*(N)  CS, CCLVAL
...
CS =  CCLVAL (I)
```

Input Arguments

I — If I > 0, item index.

If I = 0, CCLVAL assumes that I = ILOAD+1 = INEXT, *i.e.*, the "next item to load."

Function Return

CCLVAL — If the I-th item is of character string type and is of length M ≤ N, CCLVAL returns its value in the first M characters, and the remaining N-M ones are blankfilled. If N < M, the returned value is truncated to the first N characters.

If the I-th item is of numeric type, or if I exceeds the total number of items, CCLVAL returns a blank.

Procedure

Check argument I; if zero, replace as indicated; set function value as indicated. Before returning, if (1 < I < ITEMS) set the load pointer ILOAD to I, and adjust INEXT accordingly.

EXAMPLE 6.1

If the last user command starts with a keyword whose first four characters are SOLV, call subroutine SOLVER

```
CHARACTER*4 CCLVAL
...
IF (CCLVAL(1) .EQ. 'SOLV') THEN
   CALL SOLVER
END IF
```

§6.3 GET INDIVIDUAL DOUBLE FLOATING VALUE: DCLVAL

Entry point DCLVAL, which is invoked as a double-precision function, returns the double-precision floating-point value of a numeric CLIP item identified by its index.

Calling Sequence

```
DOUBLE PRECISION D, DCLVAL
...
D =  DCLVAL (I)
```

Input Arguments

I	If I > 0, item index. If I = 0, DCLVAL assumes that I = ILOAD+1 = INEXT, *i.e.*, the "next item to load."

Function Return

DCLVAL	If the I-th item is of numeric type DCLVAL returns its value as a double-precision floating-point number. If the I-th item is of character type, or if I exceeds the total number of items, DCLVAL returns zero.

Procedure

Check argument I; if zero, replace as indicated; set function value as indicated. Before returning, if 1 < I < ITEMS set the load pointer ILOAD to I, and adjust INEXT accordingly.

REMARK 6.2

An integer value is converted to a double-precision floating-point value; for example, 5 is returned as 5.0D+0.

EXAMPLE 6.2

Load items 4 through 8 into first 5 entries of the double-precision array DD:

```
      DOUBLE PRECISION  DD(5), DCLVAL
      ...
      DO 2000 J = 1,5
        DD(J) = DCLVAL(J+3)
2000  CONTINUE
```

§6.4 GET INDIVIDUAL SINGLE FLOATING VALUE: FCLVAL

Entry point FCLVAL, which is invoked as a real function, returns the single-precision floating-point value of a numeric CLIP item identified by its index.

Calling Sequence

```
F =  FCLVAL (I)
```

Input Arguments

I	If I > 0, item index. If I = 0, FCLVAL assumes that I = ILOAD+1 = INEXT, *i.e.*, the "next item to load."

Function Return

FCLVAL	If the I-th item is of numeric type FCLVAL returns its value as a single-precision floating-point number. If the I-th item is of character type, or if I exceeds the total number of items, FCLVAL returns zero.

Procedure

Check argument I; if zero, replace as indicated; set function value as indicated. Before returning, if 1 < I < ITEMS set the load pointer ILOAD to I, and adjust INEXT accordingly.

REMARK 6.3

An integer value is converted to a single-precision floating-point value; for example, 5 is returned as 5.0.

EXAMPLE 6.3

A three item keyword phrase has the following form

LIMITS = *r1 r2*

It is desired to load *r1* and *r2* into user-program variables XMIN and XMAX, respectively. The following code block, which assumes that LIM is the "root" of keyword LIMITS, does it:

```
IF (ICLTYP ('LIM^ITS')) .NE. 0) THEN
  XMIN = FCLVAL(0)
  XMAX = FCLVAL(0)
ENDIF
```

For example, if the actual command is

```
SET COORDINATE LIMITS = 1.5 2.57
```

then XMIN receives 1.5 and XMAX receives 2.57.

§6.5 GET INDIVIDUAL INTEGER VALUE: ICLVAL

Entry point ICLVAL, which is invoked as an integer function, returns the integer value of a numeric CLIP item identified by its index.

Calling Sequence

```
J =  ICLVAL (I)
```

Input Arguments

I	If I > 0, item index. If I = 0, ICLVAL assumes that I = ILOAD+1 = INEXT, *i.e.*, the "next item to load."

Function Return

ICLVAL	If the I-th item is of numeric type ICLVAL returns its integer value. If the I-th item is of character type, or if I exceeds the total number of items, ICLVAL returns zero.

Procedure

Check argument I; if zero, replace as indicated; set function value as indicated. Before returning, if 1 < I < ITEMS set the load pointer ILOAD to I, and adjust INEXT accordingly.

REMARK 6.4

A floating-point value is converted to an integer value following the usual FORTRAN 77 truncation procedure; for example 5.8 is returned as 5. If you prefer rounding to the next integer, use NCLVAL (§6.6).

EXAMPLE 6.4

Load items 13 to 20 into the first 8 entries of integer array IV.

```
      INTEGER  IV(30)
      ...
      DO 2000 J = 1,8
        IV(J) = ICLVAL(J+5)
2000  CONTINUE
```

§6.6 GET NEAREST INTEGER VALUE: NCLVAL

Entry point NCLVAL, which is invoked as an integer function, returns the nearest integer value of a numeric CLIP item identified by its index.

Calling Sequence

```
J = NCLVAL (I)
```

Input Arguments

I	If I > 0, item index. If I = 0, NCLVAL assumes that I = ILOAD+1 = INEXT, *i.e.*, the "next item to load."

Function Return

NCLVAL	If the I-th item is of numeric type NCLVAL returns the value of the nearest integer. If the I-th item is of character type, or if I exceeds the total number of items, NCLVAL returns zero.

Procedure

Similar to ICLVAL, but use the FORTRAN 77 function NINT to convert floating point to integer.

EXAMPLE 6.5

Consider the command

```
CONVERT 1.2 4.55
```

Then

```
ICLVAL (2)      returns   1
NCLVAL (2)      returns   1
ICLVAL (3)      returns   4
NCLVAL (3)      returns   5
```

§6.7 GET SINGLE-PRECISION COMPLEX VALUE: XCLVAL

Entry point XCLVAL, which is invoked as a complex function, returns the single-precision floating value of a pair of numeric CLIP item identified by index of the first one.

Calling Sequence

```
COMPLEX X, XCLVAL
...
X =  XCLVAL (I)
```

Input Arguments

I — If I > 0, index of the first item.

If I = 0, XCLVAL assumes that I = ILOAD+1 = INEXT, *i.e.*, the "next item to load."

Function Return

XCLVAL — If both items are of numeric type XCLVAL returns their value as the real and imaginary parts of a single-precision complex number.

If an item is not of numeric value, or is out of range, the corresponding component is set to zero. If both items are nonnumeric or out of range, both components are set to zero.

Procedure

Check argument I; if zero, replace as indicated; set function value as indicated. Before returning, if 1 < I < ITEMS - 1 set the load pointer ILOAD to I+1, and adjust INEXT accordingly.

REMARK 6.5

Any integer value is converted to single-precision floating point as usual.

REMARK 6.6

A reference to XCLVAL is equivalent to two successive FCLVAL calls with the first value going to the real part and the second going to the imaginary part.

EXAMPLE 6.6

The last command is

```
SET DAMPING COEFFICIENT = 0.0349, (-1/40)
```

Then the following code

```
COMPLEX GAMMA, XCLVAL
...
GAMMA = XCLVAL (4)
```

stores the complex value (0.0345,-0.025) into variable **GAMMA**. (The comma after **0.0345** is not strictly necessary, but cannot hurt.) On exit, **ILOAD** is **5**.

§6.8 GET INDIVIDUAL DOUBLE-PRECISION COMPLEX VALUE: ZCLVAL

Entry point ZCLVAL, which is invoked as a double complex function, returns the double-precision complex value defined by a pair of numeric CLIP items identified by the index of the first one.

Calling Sequence

```
DOUBLE PRECISION COMPLEX Z, ZCLVAL
...
Z =  ZCLVAL (I)
```

(Type declaration may be machine-dependent; see Remark below).

Input Arguments

I	If I > 0, index of first item in pair. If I = 0, ZCLVAL assumes that I = ILOAD+1 = INEXT, *i.e.*, the "next item to load."

Function Return

ZCLVAL	If both items are of numeric type ZCLVAL returns their value as a double-precision complex number. If one of the items is nonnumeric or is outside the range, zero is returned for that component. If both items are nonnumeric or out of range, both components are set to zero.

Procedure

Check argument I; if zero, replace as indicated; set function value as indicated. Before returning, if 1 < I < ITEMS set the load pointer ILOAD to I+1, and adjust INEXT accordingly.

REMARK 6.7

This data type is not part of standard FORTRAN 77 and so it may not be provided by some compilers. For such machines this entry point is undefined. Even if provided, the syntax for declaring it may vary. On byte-oriented machines, it's usually COMPLEX*16.

REMARK 6.8

Integer values, if any, are converted to double-precision floating-point values in the usual way.

REMARK 6.9

The same results may be obtained by two successive calls to DCLVAL with the first value going to the real part and the second value going to the imaginary part.

7

Loading Item Lists

§7.1 GENERAL DESCRIPTION

To load an item list with one call you use entry points named CLVAL*x*, where the last letter identifies the data type of the array that will *receive* the values: C for character, D for double-precision, F for single-precision floating, I for integer, and N for nearest integer. Entry points for loading complex arrays are not presently provided, but may be added in the future if there is sufficient demand.

Item-list processing occurs less frequently than individual item processing. Thus, Processor developers usually learn the entry points of Section 6 first. An easy way to remember the names of list-loading entry points is to take the first letter of the corresponding function entry and append it to what's left; for example, ICLVAL becomes CLVALI.

REMARK 7.1

The old (1979 vintage) list-loading entry points were named CLOAD*x*. These are still supplied but should be regarded as obsolete and replaced by CLVAL*x* in new software. The CLVAL*x* entry points are designed to work efficiently in conjunction with processing of lists after qualifiers and allow loading of comma-less lists.

§7.2 LOAD ITEM LIST: CLVALx

There are four list-loading entry points of the form CLVALx. The last letter x is C, D, F, I or N and designates the data type of the *receiving* array.

Loading starts at the "next item to load" position and is incremented on exit by the number of values loaded. The loading process terminates when one of the following exit conditions holds true:

1. The list terminates.
2. The end of the Decoded Item Table is reached.
3. A maximum number of values (specified as argument) is reached.

Calling Sequence

```
CALL   CLVALx (OPTS, M, A, N)
```

Input Arguments

OPTS — A character array containing list-interpretation option letters. The following letters are presently meaningful.

B: Accept blanks and commas as item list connectors. If omitted, only commas are considered as connectors and a blanks-only separator is interpreted as a list terminator.

E: Load list only if first item is preceded by an equals sign. This option is mandatory to load *qualifier lists* following a reference to ICLSEQ. If E is specified and no equals sign is found, nothing is loaded and argument N returns zero.

The default OPTS = ' ' is appropriate for non-qualified lists; equal signs are ignored and commas are required connectives.

M — The absolute value of M is the maximum number of values that may be loaded into A. (This is usually the array dimension.)

If M is negative, array A is initialized on entry to the list-load subroutine. Initialization means blankfilling if A is of type character, or zerofilling otherwise.

Output Arguments

A — Array that will receive the item values in the first N locations. The data type of A should correlate with the last entry-point name letter as follows:

x = C, if A is character.

x = D, if A is double-precision floating.

x = F, if A is single-precision floating.

x = I or N, if A is integer.

N — Number of values loaded into A. May be zero.

Description

On entry, initialize array A if M is negative. Clear N. If option letter E appears, exit if no equals sign precedes the first list item. Load list items until one of the exit conditions listed above is verified. As each item is loaded, increment N and ILOAD by one.

REMARK 7.2

To load K single-precision complex values, set M = 2*K (or M = -2*K if you want initialization), and call CLVALF; the number of complex items loaded should be N/2. For double-precision complex do the same with CLVALD. This should work as long as the FORTRAN compiler (a) does not check data types across subroutine interfaces, and (b) stores real and imaginary part in consecutive locations.

REMARK 7.3

For loading characters CLVALC uses the passed length of the entries of A.

EXAMPLE 7.1

Assume that the last loaded command has been

```
SELECT FREEDOMS = TX, TY, TZ
```

and that the receiving character array is IDOF(6)*4. To load the list following keyword **FREEDOMS** into IDOF, one may use

```
IF (ICLSEK('FREE') .NE. 0) CALL CLVALC (' ', 6, IDOF, N)
```

This sets IDOF(1) = 'TX', IDOF(2) = 'TY', IDOF(3) = 'TZ', and N = 3. The last three entries of IDOF are not touched; if you want them blankfilled, set the second argument of CLVALC to -6.

EXAMPLE 7.2

The current command contains only a list of eight floating point numbers *not separated by commas.*

```
3.4  -4.53  0.23  (5/3)  -8.34  7.1  0.67  (-2/7)
```

Allowing commas to be omitted is a bad programming practice, but it may happen (remember Ben Franklin's "experience is a dear school, but fools will learn at no other"). These values are to be loaded into a double-precision array dimensioned DD(24), which is to be cleared on entry to CLVALD:

```
CALL  CLSLOP (0)
CALL  CLVALD ('B', -24, DD, N)
```

The CLSLOP call sets the load pointer ILOAD to zero so that list loading will start with the first item; this is not required if the command has just been retrieved via CLREAD, but it can't hurt. On exit, DD(1) = 3.4, DD(2) = -4.53, ... and N = 8.

EXAMPLE 7.3

Assume that the current command is

```
FACTOR XK /RANGE=21,629 /PDCHEK
```

Load the two integers following qualifier **RANGE** into the 2-word integer array **IRANGE**:

```
IF (ICLSEQ('RANGE') .NE. 0) CALL  CLVALI ('Q', 2, IRANGE, N)
```

On exit, **IRANGE(1) = 21**, **IRANGE(2) = 629**, and N = 2. Note that if the comma had been omitted, as in

```
FACTOR XK /RANGE=21 629 /PDCHEK
```

only 21 would be loaded into **IRANGE(1)** and N = 1. Item **629** is here considered to be "disassociated" from the qualifier **RANGE**.

§7.3 OBSOLETE ENTRY POINTS: CLOAD*x*

Earlier versions of CLIP provided the three list-loading entry points **CLOAD*x***, where ***x*** **is C, F** or I. **CLOADF** handles both single-and double-precision floating-point lists. Loading may start at a specified item number, or be defaulted to the load-pointer.

Calling Sequence

```
CALL   CLOADx (I, M, K, A, N)
```

Input Arguments

I	If I > 0, index of item at which load is to start. If I = 0, assume that I = ILOAD+1 = INEXT.
M	Same meaning as for CLVAL*x*.
K	For CLOADC, number of characters per item stored in each entry of A. For CLOADF, set K = 1 if receiving array is double precision, else K= 0. Ignored for CLOADI

Output Arguments

A	Array that will receive the item values in the first N locations. The data type of A should correlate with the last entry-point name letter as follows: *x* = C, if A is character. *x* = F, if A is single- or double-precision floating point. *x* = I, if A is integer.
N	Number of values loaded into A. May be zero.

8

Loading Keywords and Qualifiers

§8.1 GENERAL DESCRIPTION

Two entry points, CLOADK and CLOADQ, are provided for "collecting" keywords and qualifiers, respectively, in one pass and storing them in a specified character array. These entry points differ from ICLSEK and ICLSEQ in that no search for specific strings is made. Thus, use of CLOADK and CLOADQ is appropriate when the Processor is to perform its own matching.

In the present version of the document, four more functions have been added: CCLKEY, CCLQUL, ICLNKY and ICLNQL. These entry points are intended to work in conjunction with ICLKYP and ICLQLP, which are documented in §5.

§8.2 LOAD KEYWORDS LIST: CLOADK

Subroutine CLOADK scans the Decoded Item Table for keywords, starting at the "next item to load" position. All keywords found and their indices are stored in a character array specified in the argument list. The keyword indices (in the Decoded Item Table) may be optionally returned in an integer array provided for this effect.

Calling Sequence

```
CALL   CLOADK (OPTS, M, A, L, N)
```

Input Arguments

OPTS — A character array containing *uppercase* option letters. Presently the only option implemented is

L: Return indices of matched keywords in argument L. If L does not appear, the fourth argument is a dummy argument.

M — The absolute value of M is the maximum number of values that may be loaded into A. (This is usually the array dimension.)

If M is negative, array A is initialized with blanks on entry to the subroutine.

Output Arguments

A — Array that will receive the keywords found by CLOADK.

L — If option letter L appears in OPTS, integer array that will receive the indices of the keywords found by CLOADK. In other words, L(i) receives the Decoded Item Table index of A(i) for i = 1, ... N.

If the option is not exercised, this is a dummy argument and an integer zero may be inserted in the calling sequence.

N — Number of values loaded into A. May be zero.

Procedure

On entry, initialize array A if M is negative. Clear N. Scan for keywords starting at INEXT. If a keyword is found, store it in array A and increment N; if option letter L is specified, store the index into array L.

REMARK 8.1

CLOADK does not resolve the ambiguity noted in §5.1 unless the first item of a character list is always preceded by an equals sign.

EXAMPLE 8.1

Assume that the last loaded command is

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE
```

and that **ILOAD** = 0. To load all keywords into character array **KEYS*4(8)**, do

```
CALL CLOADK (' ', 8, KEYS, 0, N)
```

On return from **CLOADK**, **KEYS(1)** = **'SOLV'**, **KEYS(2)** = **'FOR'**, **KEYS(3)** = **'X'**, and **N** = **3**. The remaining entries of **KEYS** are not altered. Note that the first keyword is truncated to four characters because that is the passed character length of **KEYS**.

§8.3 LOAD QUALIFIER LIST: CLVALQ

Subroutine CLOADQ scans the Decoded Item Table for qualifiers, starting at the "next item to load" position. All qualifiers found are stored in a character array specified in the argument list. The qualifier indices may be optionally requested.

Calling Sequence

```
CALL   CLOADQ (OPTS, M, A, L, N)
```

Input Arguments

OPTS — A character array containing *uppercase* option letters. Presently the only option implemented is

L: Return indices of matched keywords in argument L. If L does not appear, the fourth argument is a dummy argument.

M — The absolute value of M is the maximum number of values that may be loaded into A. (This is usually the array dimension.)

If M is negative, array A is initialized with blanks on entry to the subroutine.

Output Arguments

A — Array that will receive the keywords found by CLOADQ.

L — If option letter L appears in OPTS, integer array that will receive the indices of the qualifiers found by CLOADK. In other words, L(i) receives the Decoded Item Table index of A(i) for i = 1, ... N. If the option is not exercised, this is a dummy argument and an integer zero may be inserted in the calling sequence.

N — Number of values loaded into A. May be zero.

Procedure

On entry, initialize array A if M is negative. Clear N. Scan for qualifiers starting at INEXT. If a qualifier is found, store it in array A and increment N; if option letter L has been specified, store the index in array L.

EXAMPLE 8.2

Assume that the last loaded command is

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE
```

and that ILOAD = 0. To load all qualifiers into character array QUALS*6(4), do

```
CALL CLOADQ (' ', 4, QUALS, 0, N)
```

On return from CLOADQ, QUALS(1) = 'RANGE', QUALS(2) = 'SCALE', and N = 2.

§8.4 GET KEYWORD GIVEN ITS POSITION: CCLKEY

Character function CCLKEY returns the value of a keyword given the keyword position as argument.

Calling Sequence

```
CHARACTER*(n) KEY, CCLKEY
...
KEY =   CCLKEY (IK)
```

Input Arguments

IK — The keyword position (do not confuse it with its item index).

Function Return

CCLKEY — Left justified value of the IK^{th} keyword if one exists, otherwise it is blank. If the keyword length exceeds the passed length, the rightmost characters will be truncated.

Procedure

The Decoded Item Table is scanned from the beginning while counting keywords. When the count reaches IK, the keyword value is returned. Otherwise, a blank is returned.

EXAMPLE 8.3

Assume that the last loaded command is

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE STORE
```

Then

```
CHARACTER*8 KEY, CCLKEY
...
KEY = CCLKEY (4)
```

places 'STORE' into KEY because STORE is the fourth keyword (the first three are SOLVE, FOR, and X).

§8.5 GET QUALIFIER GIVEN ITS POSITION: CCLQUL

Character function CCLQUL returns the value of a qualifier given the qualifier position as argument.

Calling Sequence

```
CHARACTER*(n) KEY, CCLQUL
...
KEY =   CCLQUL (IQ)
```

Input Arguments

IQ	The qualifier position (do not confuse it with its item index).

Function Return

CCLQUL	Left justified value of the IQ^{th} qualifier if one exists, otherwise it is blank. If the qualifier length exceeds the passed length, the rightmost characters will be truncated.

Procedure

The Decoded Item Table is scanned from the beginning while counting qualifiers. When the count reaches IQ, the qualifier value is returned. Otherwise, a blank is returned.

EXAMPLE 8.4

Assume that the last loaded command is

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE STORE
```

Then

```
CHARACTER*8 KEY, CCLQUL
...
KEY = CCLQUL (4)
```

places **'SCALE'** into KEY because SCALE is the second qualifier.

§8.6 GET NUMBER OF KEYWORDS: ICLNKY

Integer function ICLNKY, which is called with an empty argument, returns the number of keywords in the last command.

Calling Sequence

```
NK =    ICLNKY ()
```

Function Return

ICLNKY	Number of keywords in the last command. May be zero.

Procedure

The Decoded Item Table is scanned from beginning to end while counting keywords. The count is returned as function value.

EXAMPLE 8.5

Assume that the last loaded command is

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE STORE
```

Then ICLNKY() returns 4, which is the number of keywords (**SOLVE**, **FOR**, **X** and **STORE**).

§8.7 GET NUMBER OF QUALIFIERS: ICLNQL

Integer function ICLNQL, which is called with an empty argument, returns the number of qualifiers in the last command.

Calling Sequence

```
NK =   ICLNQL ()
```

Function Return

ICLNQL	Number of qualifiers in the last command. May be zero.

Procedure

The Decoded Item Table is scanned from beginning to end while counting qualifiers. The count is returned as function value.

EXAMPLE 8.6

Assume that the last loaded command is

```
SOLVE FOR X=(1/3) /RANGE= 25,86 /SCALE STORE
```

Then **ICLNQL()** returns 2, which is the number of qualifiers (**RANGE** and **SCALE**).

THIS PAGE LEFT BLANK INTENTIONALLY.

9

Retrieving Item Information

§9.1 GENERAL DESCRIPTION

This Section describes some function entry points by which miscellaneous information about items in the Decoded Item Table can be directly retrieved. For example, prefix, separator, item type code, and total number of items.

From a Processor developer's standpoint, the most useful entry point is possibly ICLTYP, which returns item data type codes.

REMARK 9.1

These entry points are useful for detailed processing of commands that do not quite fit the standard CLAMP format of Section 3.

§9.2 RETRIEVE ITEM PREFIX: CCLPRE

Entry CCLPRE, referenced as a CHARACTER*1 function, returns the prefix character associated with an item identified by its index in the Decoded Item Table.

Calling Sequence

```
CHARACTER*1    CH, CCLPRE
...
CH =  CCLPRE (I)
```

Input Argument

I	If I > 0, item number. If zero, INEXT is assumed.

Function Return

CCLPRE	Item prefix (see §4.1). If argument is out of bounds, a blank is returned.

Procedure

Set CCLPRE to blank. Check whether argument is in bounds; if so fetch prefix character from Decoded Item Table and return.

REMARK 9.2

The only prefix the Processor is normally interested in is the qualifier prefix, which is the slash by default. The qualifier prefix may be retrieved by calling CLCHAR as described in §11.1.

EXAMPLE 9.1

Assume that the last command is

```
OPEN /ROLD 3, [REAGAN]BUDGET.DEF /LIMIT=INFINITE
```

Then CCLPRE(2) returns /, which is the prefix of qualifier ROLD.

§9.3 RETRIEVE ITEM SEPARATOR: CCLSEP

Entry point CCLSEP, which is referenced as a CHARACTER*1 function, returns the separator associated with an item identified by its index in the Decoded Item Table.

Calling Sequence

```
CHARACTER*1    CH, CCLSEP
...
CH =  CCLSEP (I)
```

Input Argument

I	If I > 0, item number. If I = 0, INEXT is assumed.

Function Return

CCLSEP	Item separator (cf. §4.1). If the argument is out of bounds, a blank is returned.

Procedure

Set CCLSEP to blank. Check whether argument is in bounds; if so fetch prefix character from Decoded Item Table and return.

REMARK 9.3

For command processing the most interesting nonblank separators are the equals sign, which separates a keyword or qualifier from assigned values, and the comma, which connects items that pertain to an item list.

EXAMPLE 9.2

Assume that the last command is

```
OPEN /ROLD 3, [REAGAN]BUDGET.LIB /LIMIT=1500000
```

Then CCLSEP(3) returns a comma, which follows integer 3, and CCLSEP(5) returns an equals sign, which follows qualifier LIMIT.

§9.4 RETRIEVE LIST LENGTH: ICLIST

Integer function ICLIST returns the length of a item list that starts at a specified index.

Calling Sequence

```
LL =  ICLIST (I)
```

Input Argument

I	If I > 0, item number. If I = 0, INEXT is assumed.

Function Return

ICLIST	List length. If I is in range, the length will always be one or greater (because a isolated item is an one-item list). If I is out of range, ICLIST returns zero.

§9.5 RETRIEVE NUMBER OF ITEMS: ICLNIT

Integer function ICLNIT, which is called with an empty argument, returns the total number of items in the Decoded Item Table.

Calling Sequence

```
ITEMS = ICLNIT ( )
```

Function Return

ICLNIT	The total number of items in the Decoded Item Table.

EXAMPLE 9.3

Write a two-line Processor code block that prints the data type codes of all items in the Decoded Item Table. Here it is:

```
      DO 2000  I = 1,ICLNIT()
2000    PRINT *, 'Data type of item number',i,' is ',ICLTYP(I)
```

Function **ICLTYP** is described in §9.7.

§9.6 RETRIEVE LOAD POINTER: ICLOAD

Integer function ICLOAD, which is called with an empty argument, returns the current value of the load pointer ILOAD.

Calling Sequence

```
ILOAD = ICLOAD ( )
```

Function Return

ICLOAD	The current value of the load pointer.

EXAMPLE 9.4

A message call usually resets ILOAD. Assuming that the message contains only directives, write a code block that restores ILOAD.

```
ISAVE = ICLOAD ( )
CALL  CLPUT ( ...  )
CALL  CLSLOP (ISAVE)
```

Subroutine CLSLOP is described in §10.3.

§9.7 RETRIEVE ITEM TYPE: ICLTYP

Integer function ICLTYP returns the data type code of an item identified by its index in the Decoded Item Table. (This is the actual code stored in the Table.)

Calling Sequence

```
ITYPE = ICLTYP (KEY)
```

Input Argument

I	If I > 0, item index.
	If I = 0, INEXT is assumed.

Function Return

ICLTYP	Returns the item type code:
	ICLTYP = $n > 0$ if item is n-character string.
	ICLTYP = 0 if item is integer.
	ICLTYP = -1 if item is floating-point (which is always stored in double-precision form).
	If the argument is out of bounds, ICLTYP returns zero.

Procedure

Initialize ICLTYP to zero. An argument-in-bounds check is made and if verified the stored type code is fetched into ICLTYP.

EXAMPLE 9.5

Assume that the last command is

```
COORDINATES NODE 1 = 2.5, (5/3), -5.4
```

Then the references ICLTYP(1), ICLTYP(3) and ICLTYP(4) return 11, 0 and -1, respectively.

10

Miscellaneous Operations

§10.1 GENERAL DESCRIPTION

This Section describes entry points for command-related operations that do not fit the framework of the previous Sections.

§10.2 GET ERROR INFORMATION: CLEINF

Entry point CLEINF returns error information on a specific error condition. CLIP detected errors (and in general all NICE errors) are characterized by a 4-character key, an associated integer code, and an explanatory message. Given the integer code, CLEINF may be used to retrieve the message, or given the error key, CLEINF may be used to get the error code. The type of operation is defined by a selector argument.

Calling Sequence

Three possible calling sequences are:

```
CALL  CLEINF ('C', ICODE, EKEY, 0)
CALL  CLEINF ('M', ICODE, MSG, KCH)
CALL  CLEINF ('T', ICODE, EKEY, 0)
```

The first form is used to get ICODE given EKEY; the last argument is a dummy one. The second form is used to retrieve MSG(1:KCH) given ICODE. The third form, which returns EKEY given ICODE, is used primarily in code testing.

Input Arguments

ICODE	Error code. An input argument if the first argument is M or T.
EKEY	Four-letter error key. An input argument if the first argument is C.

Output Arguments

ICODE	Output argument if first argument is C.
EKEY	Output argument if first argument is T.
MSG	Error message. Output argument if first argument is M.
KCH	Number of characters returned in MSG. Output argument if first argument is M.

Procedure

CLEINF simply calls NEKINF, which handles error information retrieval for the entire NICE system and resides on the NICE utilities file. Users interested in the inner details should study the source code of NEKINF.

REMARK 10.1

This entry point is primarily intended for those Processor developers that intend to do their own error handling.

§10.3 GET ERROR COUNTS: CLERR1

Entry point CLERR1 returns error count information.

Calling Sequence

```
CALL  CLERR1 (KFERR, KWERR)
```

Output Arguments

KFERR	Count of fatal errors detected by CLIP since Processor execution start.
KWERR	Count of warning errors detected by CLIP since Processor execution start.

Procedure

Simple access to an internal common block to pick up the value of the counters.

§10.4 GET LAST IMAGE: CLGLIM

Entry point CLGLIM returns to the calling program the last command image loaded by CLREAD. This is similar to what CLGET returns as third argument, but is intended for programs that make use of CLREAD to get parsed commands. Its main application is the processing of error conditions in programs that absorb voluminous input data.

Calling Sequence

```
CALL  CLGLIM (IMAGE)
```

Output Argument

IMAGE	A character string into which CLGLIM places the image of the last ordinary command processed by CLREAD.

Procedure

Access dataline collector and retrieve portion marked "to be discarded" (the processed command image is not immediately erased, as §4.2 would make you believe). Store this text into argument IMAGE and return.

REMARK 10.2

The main use of CLGLIM is for displaying input images after an error has been detected by a Processor that uses CLREAD to read commands.

EXAMPLE 10.1

The last command read by a material Processor is

```
ELASTIC MODULUS = -3.7E6
```

The Processor logic complains about a negative elastic modulus. After printing an appropriate error message, it branches to a subroutine GUILTY that displays the image that caused the error:

```
SUBROUTINE GUILTY
CHARACTER*80 IMAGE
CALL CLGLIM (IMAGE)
PRINT '(A/1X,A)', ' Data line in error:',image(1:LENETB(image))
RETURN
END
```

Function LENETB is described in Appendix D. For this use an 80-character image is enough, as it doesn't matter too much if truncation occurs.

§10.5 SHOW LAST IMAGE: CLSLIM

Subroutine CLSLIM writes the last command image to the bulk print file if the dataline echo is off; otherwise, nothing happens.

Calling Sequence

```
CALL  CLSLIM
```

Procedure

If the dataline echo mode is on, exit. Otherwise proceed as for CLGLIM (§10.1) but instead of storing the last command image to an argument, write it to the bulk print file.

§10.6 SET LOAD POINTER: CLSLOP

Subroutine CLSLOP sets the load pointer ILOAD to the value specified in the argument.

Calling Sequence

```
CALL  CLSLOP (I)
```

Input Argument

I	The value to be assigned to the load pointer. If less than zero, I = 0 is assumed. If greater than ITEMS, I = ITEMS is assumed.

THIS PAGE LEFT BLANK INTENTIONALLY.

11

Retrieving Run Information

§11.1 GENERAL DESCRIPTION

This section collects entry points that provide miscellaneous information about the run state and certain run parameters maintained by CLIP. The information is not related to a specific command or command items.

§11.2 GET CONTROL CHARACTER: CLCHAR

Entry point CLCHAR, which is referenced as a character function, returns the character used by CLIP for certain control functions. The type of function is identified by the argument.

Calling Sequence

```
CALL CLCHAR (KEY, CH)
```

Input Argument

KEY — One of the keywords specified in Table 11.1. Only the first four characters are considered significant.

Output Argument

CH — Control character associated with the function specified in KEY. The default value shown in Table 11.1 is returned unless the value has been changed through a SET CHARACTER directive.

If KEY is not matched, a blank character is returned.

Table 11.1 **Information Returned by CLCHAR**

Argument key	*Information Returned*	*Default*
ARGBEG	Left formal-argument delimiter in procedure body	[
ARGEND	Right formal-argument delimiter in procedure body	]
DIRPRE	Directive prefix	*
ENDSRC	End-of-command-source sentinel	@
EOL1	End of line terminator #1 (also comment sentinel #1)	.
EOL2	End of line terminator #2 (also comment sentinel #2)	$
MACBEG	Left macrosymbol delimiter	<
MACEND	Right macrosymbol delimiter	>
MACPAR	Macro definition parameter marker	%
QUAPRE	Qualifier prefix	/
REPEAT	Item repetitor	@

§11.3 GET RUN STATE DATA: ICLRUN

You may interrogate CLIP on run state or run parameters by calling the integer function ICLRUN. The type of information you seek is specified in the argument.

Calling Sequence

```
INF = ICLRUN (KEY)
```

Input Argument

KEY	One of the information retrieval keys listed in Table 11.2.

Function Return

ICLRUN	Returns the information indicated in Table 11.2. If the argument key is not matched, ICLRUN returns zero.

Table 11.2 **Information Returned by ICLRUN**

Argument key	*Information Returned*
RUN	Batch/interactive indicator. In general, 0 : run is batch mode. >0 : run is interactive. Under VAX/VMS, further details are available: 1: command procedure executed interactively 2: conversational (terminal input). 10,11,12: as above, but spawned process.
LIW	Line input width in characters (normally 80 characters, but may be expanded up to 132)
LPW	Line print width in characters (normally 80 in interactive mode and 132 in batch mode)

§11.4 GET LOGICAL UNIT DATA: ICLUNT

Integer function ICLUNT returns the logical unit number of certain card-image files used by CLIP.

Calling Sequence

```
LU =  ICLUNT (KEY)
```

Input Argument

KEY	One of the information retrieval keys listed in Table 11.3.

Function Return

ICLUNT	Returns the information indicated in Table 11.3. If the argument key is not matched, ICLUNT returns zero.

Table 11.3 **Information Returned by ICLUNT**

Argument key	*Information Returned*	*Usual Value*
CIN	Logical unit number of the command source file from which CLIP is reading data lines. If zero, the default input device is assumed.	0
ERR	Logical unit number of the error print file if greater than zero. If zero, error messages go to the default print file (the terminal in interactive mode).	0
LOG	Logical unit number of the command log file if one is currently open, otherwise zero.	0
PRT	Logical unit number of the bulk print file.	6

12

Retrieving Macrosymbol Values

§12.1 GENERAL DESCRIPTION

The processor may retrieve the value of a macrosymbol or of a general expression that contains macrosymbols through function entry points of the form *x*CLMAC. The first letter identifies the data type of the function return: C for character, D for double-precision, F for single-precision floating-point, I for integer, and N for nearest integer. The function argument is a character string that carries the macrosymbol or macro expression.

REMARK 12.1

These entry points are for very advanced developers, who are expected to be thoroughly familiar with the macrosymbol facilities of CLIP as described in Volume II.

§12.2 GET MACRO VALUE: *x*CLMAC

There are five macro-value-retrieval entry points of the form *x*CLMAC, which are referenced as functions. The first letter *x* is C, D, F, I or N and designates the data type of the returning value. There is a single argument: a character string that has the macrosymbol or macroexpression to be evaluated.

Calling Sequence

```
V = xCLMAC (TEXT)
```

Input Argument

TEXT — A character string that has the name of the macrosymbol or the text of the macroexpression to be evaluated.

For example, FCLMAC('pi') returns <pi>= π = 3.14159165... in FCLMAC, and DCLMAC('exp(<pi>') returns <exp(<pi>)>= e^{π} in DCLMAC.

As the examples show, the outer pair of < > delimiters may be omitted.

The length of TEXT should not exceed 480 characters, which is fairly generous.

Function Return

*x*CLMAC — The value of the argument expression, returned in the data type specified by the first letter of the function name:

x = C, character string (passed length assumed).

x = D, double-precision floating-point.

x = F, single-precision floating-point.

x = I or N, integer.

Description

On entry, surround argument text with a < > pair, prefix the whole by the evaluate-directive key *VAL and submit to self as a one-line message. The expression that follows *VAL is processed by the macrosymbol facility. Access the result and return as function.

REMARK 12.2

If the result of the argument evaluation is a character string, DCLMAC, FCLMAC, ICLMAC and NCLMAC return zero.

REMARK 12.3

If the result of the argument evaluation is numeric, CCLMAC returns blank.

EXAMPLE 12.1

`CCLMAC ('ifdef(<exp>;true;false')`	*returns*	**TRUE**
`FCLMAC ('2^.5*<exp(-2)>')`	*returns*	$\sqrt{2}/e^2$
`ICLMAC ('max(12;<pi>*<pi>)')`	*returns*	**12**

13

Workpool Manager Interface

§13.1 GENERAL DESCRIPTION

The NICE system now contains a local data manager called the Workpool Manager, or WM. CLIP communicates with WM through a directive/macrosymbol interface as documented in §9 of Volume II. The Processor may communicate with WM through the set of entry points documented here. Bypassing the directive interface cuts down the overhead involved in command item processing.

As of this writing the WM entry points are experimental and subject to frequent change. So a formal description will have to wait for the next cycle of this document.

APPENDICES

A

A $300,000 Calculator

§A.1 SIMULATING AN RPN CALCULATOR

To illustrate the simplest form of commands, we are going to build a "virtual calculator" on the VAX. More specifically, a "toy processor" that simulates an RPN (Reverse Polish Notation) calculator exemplified by the popular Hewlett-Packard (HP) models.

Recall that RPN calculators are *stack machines*. We will assume a 4-register operational stack:

```
T       (top)
Z
Y
X       (bottom)
```

These registers are programmed to hold signed single-precision floating-point numbers. We shall use parentheses to denote the value stored in a stack registers; thus (X) means the contents of register X. To represent such a stack in a FORTRAN program, a labelled common block will do:

```
common /STACK/ x, y, z, t
real x, y, z, t
```

Deciding upon this representation at this early stage is not capricious. It responds to a basic design principle:

> *Design the data first*

Designing the stack for this toy processor takes perhaps ten seconds. For the example Processor of Appendix B, data design takes a couple of hours. For a real-life NICE Processor that has to communicate with other Processors, it may take several weeks. Whatever it takes, it's time well spent.

The Basic Commands

We shall assume that each calculator command can have only *one* item. Plainly the ultimate in command language simplicity!

The item can be either a number (written as either integer or floating-point number, it doesn't matter) or one of the following keywords:

```
ON
ENTER
ADD
SUBTRACT
PS
OFF
```

The underlying idea is that each one word command simulates a keystroke on a hand-held calculator.

The keyword roots will be ON, E, A, S, P and OF, respectively, which uniquely identify the commands. (Of course, should more commands be added these roots may have to be altered; for example, if SQRT is added then root S becomes ambiguous and we must expand the root of SUB to SU.)

Operations

Typing a numeric value causes the stack to be raised (as discussed below for operation ENTER) and the value to be stored, as a floating point constant, in register X. This is true even if an integer is typed; for example typing 26 results in (X) ← 26.0.

The meaning of keyword commands is as follows.

ON	"Turns on" the calculator. All stack registers are initialized to zero. If typed during the run, it has the effect of a calculator's "CLEAR" key.
ENTER	Copies (X) into register Y and *raises* the stack (HP terminology). More precisely, (Z) → T, (Y) → Z, (X) → Y, and (T) is lost.
ADD	Adds (X) to (Y), places result in (X) and *lowers* the stack; that is, (Z) → Y, (Y) → X, while T is unchanged. The new value of X is printed.
SUB	Subtracts (X) from (Y), places result in (X) and *lowers* the stack; that is, (Z) → Y, (Y) → X, while T is unchanged. The new value of X is printed.
PS	Prints the complete stack, i.e. (X), (Y), (Z), and (T).
OFF	"Turns off" the calculator by terminating the run of the Processor.

Next we describe how these operations can be implemented as simple FORTRAN subroutines.

Turning On

The ON operation consists of a simple initialization:

```
      subroutine ON
      common /STACK/  x, y, z, t
      real x, y, z, t
      x =    0.0
      y =    0.0
      z =    0.0
      t =    0.0
      return
      end
```

Storing a Number

The code for storing a new numeric value **xnew** in register **X** is

```
      subroutine STOREX (xnew)
      common /STACK/  x, y, z, t
      real x, y, z, t
      call   ENTER
      x =    xnew
      return
      end
```

where the **ENTER** subroutine is described below.

The ENTER Operation

Implementation of the **ENTER** operation is equally straightforward:

```
      subroutine ENTER
      common /STACK/  x, y, z, t
      real x, y, z, t
      t =    z
      z =    y
      y =    x
      return
      end
```

The ADD and SUBTRACT Operation

The implementation of these two operations is quite similar:

```
      subroutine ADD
      common /STACK/  x, y, z, t
      real x, y, z, t
      x =    y + x
      y =    z
      z =    t
      print *, '   X:', x
      return
      end

      subroutine SUB
      common /STACK/  x, y, z, t
      real x, y, z, t
      x =    y - x
```

```
      y =    z
      z =    t
      print *, '   X:', x
      return
      end
```

Printing the Stack

Very simple with a list-oriented print statement:

```
      subroutine PS
      common /STACK/  x, y, z, t
      real x, y, z, t
      print *, 'Stack:', x,y,z,t
      return
      end
```

Turning Off

This is just a run stop:

```
      subroutine OFF
      stop 'That is all, folks'
      end
```

§A.2 THE EXECUTIVE

In true bottom-up program-building style, we are ready for the *pièce de resistance*: the main program that drives all those small subroutines. Like all interactive programs fitting the NICE Processor model, the driver is essentially an infinite loop that can be expressed informally as

Program HPVAX
Begin
  Do forever {
    Get next command
    Process command }
End

This may be readily implemented as a FORTRAN main program:

```
      program HPVAX
*
*     Simulating a $50 RPN calculator on a $300,000 minicomputer
*
      character*4 key, CCLVAL
      integer ICLVAL
      real x, FCLVAL
1000    continue
        call   CLREAD (' Command> ',
     $                 ' ON, ENTER, ADD, SUB, PS, OFF')
          if (ICLTYP(1) .le. 0)  then
            x = FCLVAL(1)
            call STOREX (x)
          else
            key = CCLVAL(1)
            call DOKEY (key)
          end if
        go to 1000
      end
```

Program HPVAX asks for the next command by calling CLREAD (§2.7). This call specifies

Command>

as the prompt message you will see on the terminal. The "splash" line, which will appear on the terminal if the "verbose" echo mode is turned on, is simply a remainder of the available commands.

On return from CLREAD, the program checks for the type code of the first (and only) command item through ICLTYP (§10.3). If the item is numeric, its floating-point value is retrieved through function FCLVAL (§6.3), and STOREX is called to put it in register X. Otherwise the command is a keyword, which is retrieved via CCLVAL (§6.1) and placed into character string variable key; subroutine DOKEY is called to interpret the command.

Subroutine DOKEY is essentially a six-way "case" statement:

```
      subroutine DOKEY (key)
      character*(*)  key
      logical CMATCH
      if (CMATCH (key, 'A^DD')) then
        call ADD
      else if (CMATCH (key,'E^NTER')) then
        call ENTER
      else if (CMATCH (key,'OF^F')) then
        call OFF
      else if (CMATCH (key,'ON')) then
        call ON
      else if (CMATCH (key,'P^S'))  then
        call PS
      else if (CMATCH (key,'S^UB'))  then
        call SUB
      else
        print *, '*** Illegal or ambiguous keyword: ',key
      end if
      return
      end
```

The logical function CMATCH compares two keywords following the "root + extension" rules stated in §5.1. The calling sequence is described in Appendix D.

Note that the IF-THEN-ELSE construction tests commands *alphabetically.* There is one motivation behind this: if you later come back to DOKEY to insert additional commands (and you will), having sorted keywords greatly simplifies checking whether their roots ought to be expanded to avoid ambiguities as discussed in §5.1.

The implementation of the six-command calculator is complete.

§A.3 RUNNING HPVAX ON THE VAX

If you have never run a CLIP-supported interactive program before, then HPVAX is not a bad place to start. The internal logic is so straightforward that there is little chance that the workings of the calculator will mask the goings-on of the interactive process.

> *The following material assumes that the work will be performed on a VAX 11/7xx minicomputer running under VAX/VMS.*

Preparing an Executable Image

If you do not have access to an executable image of HPVAX, you will have to make one by yourself. Here are the basic steps explained in cookbook fashion.

Since HPVAX is so tiny, it is convenient to have all of its code in a single source file, say HPVAX.FOR. Upon compiling it you have an object file called HPVAX.OBJ.

Next you must link to the NICE object library. On several VAX systems at LMSC and LaRC/CSM, this library resides on the file

```
NICE$OLB:NICE.OLB
```

where NICE$OLB is a system-wide logical name, so you can create an executable image by saying

```
LINK  HPVAX, NICE$OLB:NICE/LIB
```

On some systems there is a "shareable image" version of the NICE library, which is accessed by saying

```
LINK  HPVAX, NICE$OLB:SHARENICE/LIB
```

If SHARENICE is available, by all means use it, since it saves both link time and executable-image size (the latter drops from over 400 disk blocks — 1 disk block ≡ 512 bytes — to less than 10).

Whatever the library used, you should end up with an executable image file called HPVAX.EXE. To run this image you say, reasonably enough,

```
RUN  HPVAX
```

and now the fun begins.

Typing Commands

HPVAX's prompt message will show up on the screen as

Command>

and the cursor stays "frozen" after the angle bracket (>). It is waiting for you! You may begin by responding ON followed by a carriage-return, and the prompt reappears:

```
Command> ON
Command>
```

Next type PS followed by carriage-return. The program will then print the contents of the stack, which should be four zeros, and then come back with the usual prompt:

```
Command> PS
Stack: 0.0000E+00 0.0000E+00 0.0000E+00 0.0000E+00
Command>
```

Next you should try entering numbers. Type 1, then 2, then 3, then 4, following each number with a carriage-return. Then type PS and verify that the four numbers are in the stack.

```
Command> 1
Command> 2
Command> 3
Command> 4
Command> PS
Stack:   4.0000E+00 3.0000E+00 2.0000E+00 1.0000E+00
Command>
```

Next try some ADD and SUBTRACT commands and verify that HPVAX works as an RPN calculator should.

After you acquire some proficiency, try entering multiple commands per line. CLIP will let you do this if you separate commands with semicolons, but do not forget to put a blank before each semicolon (a blank after a semicolon is not necessary but it doesn't hurt). For example:

```
Command> ON ; 1 ; 2 ; 3 ; A ; A
```

and you should see the result X = 3+2+1 = 6.

Suggested Run Exercises

EXERCISE A.1

Command Formatting. Try entering an illegal command, *e.g.*, KILL. What happens? Enter commands in lowercase and uppercase forms. Does it matter?

EXERCISE A.2

Composite Numbers. Enter (1/3) and print the stack. Did you get what you expected, *viz.* the fraction one third? Then try entering (1-(1/3)), (2^.5), ((1/9)*(1/9)), printing the stack after each entry. Comment on what's going on. Did you realize that you have an algebraic calculator (within CLIP) embedded in a stack calculator (HPVAX)?

EXERCISE A.3

Built-in Macrosymbols. Enter <pi> and print the stack. Do you recognize that number? Then try entering <exp(1)>, <sing(45)>^2>, <log10(2)> and <atan2g(1;1)>, printing the stack after each entry. Comment. (To see all available built-in macrosymbols, you may type *SHOW MACROS/B/V, but only after you understand directives.)

EXERCISE A.4

Directives. Assuming that the source file HPVAX.FOR is in the same directory as you are running HPVAX.EXE from, say

Command> *type hpvax.for

What comes to your screen? By printing the stack before and after this peculiar command, you may verify that *nothing* has happened to HPVAX. You have just entered a ***directive***, which is a special command for internal consumption by CLIP. You can tell it apart from an ordinary command because its action verb is prefixed by an asterisk. Try *TYPE or *LIST on some other card-image files.

EXERCISE A.5

Echo Control. After two or three *TYPEs, you should be an old hand at directives. Now enter *SET ECHO = ON. Type some command; what do you see? Then *SET ECHO = BELL. Then *SET ECHO = VERBOSE. For obvious reasons, these are called *echo options.* The ordinary (default) options may be reset by entering *SET ECHO only.

EXERCISE A.6

More on Display Options. If you are working at a VT100 or VT100 compatible terminal, try *SET ECHO/PROMPT = RV and *SET ECHO/SPLASH = RV followed by *SET ECHO = VERBOSE.

Suggested Programming Exercises

EXERCISE A.7

Extending the Calculator. Implement **MULTIPLY**, **DIVIDE** and **SQUAREROOT** operations.

EXERCISE A.8

Continuing the previous exercise: How do you take care of arithmetic errors such as division by zero on **DIVIDE** or negative arguments to **SQUAREROOT**? Discuss alternatives.

EXERCISE A.9

Complex Arithmetic. Convert **HPVAX** to operate as a "complex calculator" by globally substituting all **real** declarations by **complex** and replacing **FCLVAL** by **XCLVAL**. Now each "numerical entry" involves entering a number pair. Do the two values have to be separated by a comma? And what happens if you only enter one?

EXERCISE A.10

Messages. Implement a **CLEAR** command that does the same thing as **ON**, **i.e.**, clears the stack. But instead of calling subroutine **ON**, **CLEAR** should call the following subroutine:

```
subroutine CLEAR
call  CLPUTW ('ON')
return
end
```

which *sends a message* through the put-message-and-wait entry point **CLPUTW**. Test it by filling the stack with numbers, entering **C**, then **PS**. (To see the message in action, turn the echo on before typing **CLEAR**.)

EXERCISE A.11

More on Messages. Implement a BELL command that mails the directive **'*SET ECHO = BELL'** through the immediate-message entry point **CLPUT**. How can the user turn off the bell?

EXERCISE A.12

An Embryonic Database. Conceptually design **SAVE** and **RESTORE** commands to save the stack on a permanent FORTRAN file, and to read it back. Do you think these operations are worth the effort for this application?

EXERCISE A.13

A Matrix Calculator. Suppose that each of the "stack registers" becomes an $(n \times n)$ matrix, where n is a modest number read when the calculator is turned on. Which commands would have to be changed? How should numeric values be entered? Would "matrix edit" commands be useful? Can you think of any uses for such a calculator when $n = 2$ or $n = 3$?

B

A Direct Boundary Element Processor

§B.1 BACKGROUND

The case study presented in Appendix A deals with a "toy processor" purposely chosen to illustrate the simplest possible input form: one-item commands. The whole Processor can be written and tested in a couple of hours.

The example Processor presented in this Appendix is still quite simple as production Processors go, but is no longer trivial. It requires about one week to put together. The Processor solves a two-dimensional elastostatic problem by a directly-formulated* Boundary Element Method (BEM), and is appropriately named DBEM2.

The "kernel" of the Processor is a BEM-program adapted from the book *Boundary Elements Methods in Solid Mechanics* by S. L. Crouch and A. M. Starfield, reference B-1. (See page B-53.) The program is called TWOBI and is presented in Appendix A of the book; it is based on the boundary-integral theory covered in Section 6 therein.

The program is appropriate as an example of the use of interactive techniques because the input data are fairly simple but the commands are now of multiple-item type and thus serve to illustrate things like phrases, item lists, and defaults.

---

* The term *direct formulation* refers to the technique used in deriving the governing boundary-integral equations. Direct methods are formulated from the start in terms of physical quantities such as displacement and stress fluxes. On the other hand, indirect methods are formulated in terms of source strength distributions, which have no direct physical meaning and are eventually eliminated following spatial discretization.

§B.2 THE BEM MODEL

The BEM model accepted by DBEM2 is a finite or infinite domain of elastic isotropic material under a plane-strain condition. If finite, the domain is enclosed by a boundary that consists of *line segments* as illustrated on the left of Figure 1.

Figure B.1 Two-dimensional domains that can be treated by DBEM2

If the domain is infinite, it is assumed to be the *exterior* of a cavity defined by a series of line segments. Thus Figure 1 may also be viewed as defining an exterior problem. The sense in which the boundary is traversed when the component segments are defined determines whether the problem is interior or exterior, as illustrated in the Figure.

Boundary conditions of stress-traction or displacement type may be prescribed on each segment as explained in further detail later. The prescribed values are assumed to be *constant* over each segment.

Each line segment may be discretized into one or more boundary elements. All unknown quantities (displacements or stresses) are assumed to be constant over each element. The element unknowns are evaluated at the element midpoints. There are two unknowns per element: a shear (tangential) value and a normal value; these being the conjugates of the prescribed boundary values.

The boundary unknowns are determined by solving a linear, unsymmetric system of algebraic equations. Once these unknowns are determined, stresses and displacements at any "field point" located in the interior of the domain can be readily calculated by Somigliana's superposition formula.

§B.3 THE DATA STRUCTURES

Following sound practice, we begin by designing the data structures first. The task is more complicated for DBEM2 than for the toy program of Appendix A, although it's still trivial compared to the problem of designing a "global database" shared by many Processors.

The task is simplified by the following considerations:

1. The Processor presented here is isolated from others. There is no need to transact business with a global database.
2. DBEM2 makes use of only one matrix, which is generally unsymmetric and full. There being no need to make use of sparse storage formats, an ordinary FORTRAN array suffice.
3. Everything is assumed to fit in core at one time. Not having to deal with auxiliary storage avoids many complications.

As in Appendix A, all data that has to be shared among many parts of DBEM2 are accommodated in labelled common blocks. But in the present Processor several blocks are used to group data according to function. Furthermore, the blocks are declared in separated files whose extension (on the VAX system) is `INC`. These files are inserted where they are needed via `INCLUDE` statements. The use of `INCLUDE` enforces consistency (everything is declared only once) and makes maintenance and modification much easier.

The Segment Data

We begin by setting up the data for boundary segments, which is placed in file **`SEGMENT.INC`**. The maximum number of segments is parameterized to be **`MAXSEG`**, which is set to 100 in the version listed below.

The DBEM2 user will be allowed to define segments in any order and give them arbitrary numbers from 1 through **`MAXSEG`**, so we need a "marker" array that tells which segments have been defined. We also need a counter of how many boundary elements are in each defined segment. Then there are the geometric arrays: the x and y coordinates of the end points. Finally, there are the boundary condition arrays: one integer code (related to that used by Crouch and Starfield (ref. B-1)) and two floating-point arrays of prescribed shear and normal values. Here is a list of the file that groups this information:

```
*
*     This is file SEGMENT.INC
*
      common /SEGMENT_DATA/
     $   segdef, numel, xbeg, ybeg, xend, yend, kode, bvs, bvn
      integer    MAXSEG
      parameter (MAXSEG=20)
      integer segdef(MAXSEG) ! Segment definition tag
      integer numel(MAXSEG)  ! Number of BE divisions of segment
      real    xbeg(MAXSEG)   ! X-coord of starting segment point
      real    ybeg(MAXSEG)   ! Y-coord of starting segment point
      real    xend(MAXSEG)   ! X-coord of ending segment point
```

```
      real    yend(MAXSEG)   ! Y-coord of ending segment point
      integer   kode(MAXSEG) ! Segment BC code
      real      bvs(MAXSEG)  ! Prescribed shear value
      real      bvn(MAXSEG)  ! Prescribed normal value
```

The style used in this INCLUDE file will be followed for all others. There is a COMMON declaration that lists the shared variables. Then each variable is declared on a separate line. The variable name is followed by an inline comment that provides a short description of the function of each variable. This brief documentation should be entered at the time you prepare or update the INCLUDE file.

The Material Data

Since we are dealing with a homogeneous elastic isotropic material and we ignore thermal effects, the material is fully characterized by two properties: the elastic modulus E and the Poisson's ratio ν. These two are collected in file MATERIAL.INC:

```
*
*     This is file MATERIAL.INC
*
      common /MATERIAL/  em, pr
      real    em     ! Elastic modulus
      real    pr     ! Poisson's ratio
```

The Symmetry Data

The program allows one or two lines parallel to the coordinate axes to be specified as axes of symmetry. For example, $x = 2.5$ or $y = -1.50$, or both. Three pieces of data accommodate this information: one symmetry tag (0=none, 1=symmetry about $x = a$, 2 = symmetry about $y = b$, 3 = double symmetry), and the values of a and b as appropriate. The necessary declarations are placed in file SYMMETRY.INC:

```
*
*     This is file SYMMETRY.INC
*
      common /SYMMETRY_DATA/
     $    ksym, xsym, ysym
      integer   ksym
      real      xsym, ysym
```

The Prestress Data

The program allows a constant initial-stress field to exist in the *undeformed* medium. This *prestress* tensor field is defined by the three components σ^0_{xx}, σ^0_{yy} and σ^0_{xy}. If undefined, these three values are assumed to be zero. File PRESTRESS.INC contains the appropriate declarations:

```
*
```

```
*     This is file PRESTRESS.INC
*
      common /PRESTRESS/  sxx0, syy0, sxy0
      real    sxx0   ! Prestress (initial field stress) sigma_xx
      real    syy0   ! Ibid., for sigma_yy
      real    sxy0   ! Ibid., for sigma_xy
```

Prestress data are especially important for analysis of ***unbounded domains***, for which they assume the role of conditions at infinity. For example, suppose that we want to analyze the effect of a hole in an infinite region under uniform uniaxial stress, say $\bar{\sigma}_{xx}$. Then we set $\sigma^0_{xx} = \bar{\sigma}_{xx}$, $\sigma^0_{yy} = \sigma^0_{xy} = 0$ in the input data.

The Element Data

The most voluminous data are those pertaining to the boundary elements, since typically there will be many elements per segment. The information is collected in file **ELEMENT.INC**, which reads

```
*
*     This is file ELEMENT.INC
*
      common /ELEMENT_DATA/
     $   numbe, xme, yme, hleng, sinbet, cosbet, kod, c, b, r, x
      integer    MAXELM, MAXEQS
      parameter (MAXELM=100)    ! Maximum no. of boundary elements
      parameter (MAXEQS=2*MAXELM) ! Maximum no. of discrete equations
      integer   numbe           ! Total number of boundary elements
      real      xme(MAXELM)     ! X-coor of element midpoint
      real      yme(MAXELM)     ! Y-coor of element midpoint
      real      hleng(MAXELM)   ! Half length of element
      real      sinbet(MAXELM)  ! Sine of (element,x) angle
      real      cosbet(MAXELM)  ! Cosine ibid.
      integer   kod(MAXELM)     ! Elem BC code (copies seg code)
      real      b(MAXEQS)       ! Prescribed boundary values
      real      c(MAXEQS,MAXEQS) ! Influence coefficient matrix
      real      r(MAXEQS)       ! Forcing (RHS) vector
      real      x(MAXEQS)       ! Solution vector
```

The elements arrays such as XME, YME, etc. are parameterized in terms of the maximum number of elements MAXELM.

This block also contains arrays used to set up and solve the BEM equation system, namely C, R, B and X. These are parameterized in terms of the total number of equations MAXEQS, which of course is twice MAXELM.

The Field Location Data

The final block of data pertains to the location of field points at which stresses and displacements are to be calculated once the boundary solution is obtained. The program allows these locations to be specified as equally spaced points along straight lines defined by the user. Up to MAXLIN (=100 in the version below) lines can be defined. The locations are specified by giving the x and y coordinates of the first and last points on the line, and the number of intermediate points ($\geq$0) to be "collocated" between the first and last points. An isolated point may be specified by making the first and last point coincide.

All of this information is gathered in file OUTPUT.INC:

```
*
*     This is file OUTPUT.INC
*
      common /OUTPUT_DATA/
     $   lindef, nintop, xfirst, yfirst, xlast, ylast
      integer    MAXLIN
      parameter (MAXLIN=100)
      integer   lindef(MAXLIN)  ! Line definition tags
      integer   nintop(MAXLIN)  ! No. of intermediate points on line
      real      xfirst(MAXLIN)  ! X-coor of first point on line
      real      yfirst(MAXLIN)  ! Y-coor of first point on line
      real      xlast(MAXLIN)   ! X-coor of last  point on line
      real      ylast(MAXLIN)   ! Y-coor of last  point on line
```

This concludes the design of the important data structures. Next we pass to the design of a command set to control logic of DBEM2.

§B.4 THE COMMANDS

Having described the data, we now have to design an appropriate set of commands to perform operations on the data. The writer found it convenient to chose commands headed by the following action verbs:

```
CLEAR
DEFINE
BUILD
GENERATE
SOLVE
PRINT
STOP
```

Why these particular commands? Partly from a preliminary study of the problem, and partly from the wish[es] to get several command formats so that the use of many of the entry points described in the main body of this Volume would be illustrated.

It turns out that the last wish (of illustrating various command formats) makes the command set a bit inconsistent, but that should not cause a great deal of concern. After all, it's only an example.

Another Processor developer faced with the same problem (even a simple problem like this one) may in fact come up with a radically different set of command that accomplishes virtually the same thing.

We next describe briefly what the commands do.

CLEAR	Initializes all Tables maintained by the Processor and sets some default values.
DEFINE	Enters data that are used in the definition of the problem to be solved. The DEFINE verb will be followed by another keyword that makes the data more specific.
BUILD	Indicates that the problem-definition phase is complete, and calls for the generation of the discrete governing equations. This is carried out in two phases identified by a keyword that follows BUILD.
GENERATE	Triggers the assembly of the influence coefficient matrix and force vector.
SOLVE	Triggers the solution for the unknown boundary variables.
PRINT	Prints displacements and stresses at boundary points and at specified field points.
STOP	Terminates execution of the processor.

§B.5 STARTING AT THE TOP

We are going to build the Processor Executive "top down". For this relatively small Processor it probably doesn't make much difference whether we do it top-down, bottom-up or inside-out. But adhering to this approach makes life easier for bigger Processors.

Following the top-down approach we must do the main program first. Here it is:

```
*
*     Computer Program for the Two-Dimensional Direct
*     Boundary Element Method (DBEM2)
*
*     Adapted from program TWOBI in the book Boundary Element Methods
*     Methods in Solid Mechanics by S. L. Crouch and A. M. Starfield,
*     G. Allen & Unwin, London, 1983,  by C. A. Felippa to
*     exemplify conversion to interactive operation via CLIP.
*
      program    DBEM2
*
      implicit       none
      character*8    CCLVAL, verb
      integer        ICLTYP
*
 1000   call      CLREAD (' DBEM2> ',
     $           ' CLEAR, DEFINE, BUILD, GENERATE, SOLVE, '//
     $           'PRINT, STOP')
        if (ICLTYP(1) .le. 0)   then
          print *, '*** Commands must begin with keyword'
        else
          verb =   CCLVAL(1)
          call     DO_COMMAND (verb)
        end if
        go to 1000
      end
```

Some differences with the main program of HPVAX are evident. A top-level command must start with an action verb; it cannot start with a numeric item, hence the error check. The prompt is now the name of the Processor: this is a convention followed in the NICE system.

The observant reader will note substantial similarities with the main program for Processor HPVAX presented in Appendix A. It is a fact that the top level of all Processors looks very much the same, regardless of the complexity of what lies underneath. This is not surprising if you note that all Processors fit the "do forever" model illustrated in §A.2.

The next level is DO_COMMAND, which is again a "case" statement that branches on the action verb:

```
*
```

```
*       Top level command interpreter for DBEM2
*
      subroutine     DO_COMMAND  (verb)
*
      implicit       none
      character      key*8, verb*(*)
      logical        CMATCH
*
      key =   verb
      if (CMATCH (key, 'B^UILD'))          then
        call  BUILD
      else if (CMATCH (key, 'C^LEAR'))     then
        call  CLEAR
      else if (CMATCH (key, 'D^EFINE'))    then
        call  DEFINE
      else if (CMATCH (key, 'G^ENERATE')) then
        call  GENERATE
      else if (CMATCH (key, 'H^ELP'))      then
        call  HELP
      else if (CMATCH (key, 'P^RINT'))     then
        call  PRINT
      else if (CMATCH (key, 'SO^LVE'))     then
        call  SOLVE
      else if (CMATCH (key, 'ST^OP'))      then
        call  STOP
      else
        print *, '*** Illegal or ambiguous verb: ', key
      end if
      return
      end
```

Note again that the tests are ordered so that keywords are alphabetically sorted. This makes it easier to insert new keywords without forgetting to expand roots of existing ones. For example, suppose you want to insert a PLOT command for your favorite graphic device; inserting it just before the test for PRINT makes it easy to spot that the root for the latter has to be expanded to PR.

§B.6 STARTING AND STOPPING

The CLEAR subroutine is quite simple, as it only has to zero out the model definition tables:

```
*
*     Initialize tables, set default values
*
      subroutine    CLEAR
C
      implicit      none
      include       'SEGMENT.inc'
      include       'ELEMENT.inc'
      include       'MATERIAL.inc'
      include       'SYMMETRY.inc'
      include       'PRESTRESS.inc'
      include       'OUTPUT.inc'
      integer       i
*
      do 1500  i = 1,MAXSEG
        segdef(i) = 0
        xbeg(i) =  0.0
        xend(i) =  0.0
        ybeg(i) =  0.0
        yend(i) =  0.0
        numel(i) =  0
        kode(i) =   0
        bvs(i) =   0.0
        bvn(i) =   0.0
 1500   continue
*
      do 2000  i = 1,MAXLIN
        lindef(i) = 0
 2000   continue
      numbe =  0
*
      ksym =   0
      em =     1.0
      pr =     0.0
      sxx0 =   0.0
      syy0 =   0.0
      sxy0 =   0.0
      print *, 'Tables initialized'
      return
      end
```

The function of the arrays is explained in §B.3.

Equally simple is the **STOP** subroutine:

```
*
*     Terminate the run
*
      subroutine  STOP
      stop 'Hope you enjoyed the ride'
      end
```

§B.7 DEFINING THE PROBLEM

The DEFINE command introduces problem-definition data. It is convenient to break up the definition into several types of data, which correspond closely to the data-structure grouping discussed in §B.3. Each type is identified by a keyword that immediately follows DEFINE. The keywords are:

SEGMENTS	Specifies the straight-line segments that make up the boundary of the problem to be solved.
ELEMENTS	Specifies into how many boundary elements each segment will be divided.
BOUNDARY_CONDITIONS	Specifies the boundary conditions that apply to each boundary segment.
SYMMETRY_CONDITIONS	Specifies the symmetry conditions, if any, that apply to the problem to be solved.
MATERIAL	Specifies constitutive properties of the material.
PRESTRESS	Specifies prestress data in the form of initial stress components.
FIELD	Specifies the location of field points at which displacement and stresses are to be evaluated and printed later.

Subroutine DEFINE, unlike CLEAR or STOP, branches as per the second keyword:

```
*
*     Interpret DEFINE command
*
      subroutine    DEFINE
*
      implicit      none
      character     key*8, CCLVAL*8
      integer       ICLTYP
      logical       CMATCH
*
      if (ICLTYP(2) .le. 0)  then
        print *, '*** No keyword after DEFINE'
        return
      end if
*
      key =    CCLVAL(2)
      if (CMATCH (key, 'B^OUND'))         then
        call  DEFINE_BOUNDARY_CONDITIONS
      else if (CMATCH (key, 'E^LEMENTS')) then
        call  DEFINE_ELEMENTS
      else if (CMATCH (key, 'F^IELD'))    then
        call  DEFINE_FIELD_LOCATIONS
```

```
      else if (CMATCH (key, 'M^ATERIAL')) then
        call  DEFINE_MATERIAL
      else if (CMATCH (key, 'P^RESTRESS'))   then
        call  DEFINE_PRESTRESS
      else if (CMATCH (key, 'SE^GMENTS')) then
        call  DEFINE_SEGMENTS
      else if (CMATCH (key, 'SY^MMETRY')) then
        call  DEFINE_SYMMETRY_CONDITIONS
      else
        print *, '*** Illegal or ambiguous keyword ', key,
     $           ' after DEFINE'
      end if
      return
      end
```

The program begins checking whether a keyword actually follows **DEFINE**. If so it compares them in the usual matter and calls appropriate input subroutines. These are described next.

Defining Segments

The DEFINE SEGMENT command *introduces* a series of segment-definition commands which are expected to have the form

$$\texttt{SEGMENT} = i \quad \texttt{BEGIN} = x_i^{beg}, y_i^{beg} \quad \texttt{END} = x_i^{end}, y_i^{end}$$

where x_i^{beg}, y_i^{beg} are the x, y coordinates of the starting point of the i^{th} segment, and x_i^{end}, y_i^{end} are the x, y coordinates of the ending point. The segment list is terminated by an END command that takes the control back to the main program. In listing the coordinates, the following boundary traversal convention must be observed: a closed contour is traversed in the *counterclockwise* sense if the region of interest is outside the contour (a cavity problem), and in the *clockwise* sense if the region of interest is inside the the contour (a finite body problem).

For example, to define a 4-segment boundary that encloses a square region whose corner points are (0,0), (4,0), (4,4) and (0,4), and which constitutes the region of interest, you say

```
DEFINE SEGMENTS
  SEG=1 BEGIN=0,0 END=0,4
  SEG=2 BEGIN=0,4 END=4,4
  SEG=3 BEGIN=4,4 END=4,0
  SEG=4 BEGIN=4,0 END=0,0
  END
```

(Segments may be actually defined in any order; there is also no need to number them sequentially.)

The commands that enter the segment data, plus the **END** command, are call *subordinate commands*, because they can appear if and only if the command **DEFINE SEGMENT** has been entered. The **DEFINE SEGMENT** command, which introduces the subordinate commands, is said to be the *header command* (it also goes by the names *master command*, *parent command*, *leader*, etc.)

The processing of the segment-definition commands is carried out within subroutine DEFINE_SEGMENTS:

```
*
*    Read segment-definition data
*
      subroutine    DEFINE_SEGMENTS
*
      implicit      none
      include       'SEGMENT.inc'
      character*8   key, CCLVAL
      integer       iseg, n, ICLTYP, ICLVAL, ICLSEK
      real          xy(2)
      logical       CMATCH
*
 1000   call       CLREAD  (' Segment data> ',
     $           ' Enter SEG=iseg BEG=xbeg,ybeg END=xend,yend&&'//
     $           'Terminate with END',
     $           ' ')
*
        if (ICLTYP(1) .le. 0)             then
          print *, '*** Command must begin with SEG or END'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))           then
          return
        else if (CMATCH (key, 'S^EGMENT'))  then
          iseg =   ICLVAL(2)
          if (iseg .le. 0 .or. iseg .gt. MAXSEG)  then
            print *, '*** Segment number', iseg, ' out of range'
            go to 1000
          end if
          segdef(iseg) = 1
          if (numel(iseg) .le. 0)  numel(iseg) = 1
          if (ICLSEK(3, 'B^EGIN') .ne. 0)   then
            call   CLVALF (' ', 2, xy, n)
            if (n .ge. 1)   xbeg(iseg) = xy(1)
            if (n .ge. 2)   ybeg(iseg) = xy(2)
          end if
          if (ICLSEK(3, 'E^ND') .ne. 0)   then
            call   CLVALF (' ', 2, xy, n)
            if (n .ge. 1)   xend(iseg) = xy(1)
            if (n .ge. 2)   yend(iseg) = xy(2)
          end if
        else
          print *, '*** Illegal keyword ', key,' in segment data'
```

```
      end if
      go to 1000
    end
```

The structure of this subroutine is typical of those that handle subordinate commands. A "do forever" construction is headed by a CLREAD call, and the loop is escaped only when an END command is detected. Notice the different prompt and splash-line input arguments.

This subroutine provides an example of the use of the "search for keyword" function ICLSEK described in §5.2. A keyword match is followed by a value pair retrieval through the list-loading subroutine CLVALF described in §7.2.

Note the careful handling of the case in which less than two values appear after either BEGIN or END. This facilitates *table editing*. For example, the command

```
S=3 B=45.2
```

resets XBEG(3) to 45.2; nothing else changes.

Several variations on the processing of the coordinate data are possible, and are suggested in the exercise list that appears later in this Appendix.

Digression on Subordinate Commands

Why have we used subordinate commands rather than making the user type the segment in the DEFINE command itself? Contrast the above definition of the square region with the following one:

```
DEFINE SEGMENT=1 BEGIN=0,0 END=4,0
DEFINE SEGMENT=2 BEGIN=4,0 END=4,4
DEFINE SEGMENT=3 BEGIN=4,4 END=0,4
DEFINE SEGMENT=4 BEGIN=0,4 END=0,0
```

This is not too different in terms of typing effort, so the decision for adopting a one-level and a two-level structure in terms of number of keystrokes is marginal. But note that going to a two-level scheme we have effectively separated the action of ***selecting what to define***, namely segments, from the *actual definition* by entering coordinate values. This is a key aspect of *object-oriented programming:* first *select*, then *operate*. Let us make this a command design principle:

> *Try to separate selection from operation*

If you are entering commands from a keyboard perhaps the advantages are not immediately apparent. But if you go to some form of interactive graphics input the advantages will be evident when you try to "cover" the commands through message-sending techniques. The user of such a graphic system will then see SEGMENTS in a "model definition" menu, and by pointing to it he or she is transported to another screen or window in which the process of entering the segments is actually carried out.

Defining Elements

By default, each segment contains only one boundary element (see logic of DEFINE_SEGMENT). To put more elements per segment you use the DEFINE ELEMENTS command. This introduces subordinate commands of the form

$$\texttt{SEGMENT} = i \quad \texttt{ELEMENTS} = n$$

where n is the number of boundary elements in the i^{th} segment. The data are terminated by an END command. For the square region used as an example, let's say we want 10 BEs on segments 1 and 3, and 15 BEs on segments 2 and 4:

```
DEFINE ELEMENTS
  SEG=1 EL=10 ; SEG=3 EL=10 ; SEG=2 EL=15 ; SEG=4 EL=15 ; END
```

which illustrates the fact that data may be entered in any order. The implementation shown below actually allows a more general command form:

$$\texttt{SEGMENTS} = i_1,\ldots,i_k \quad \texttt{ELEMENTS} = n_1,\ldots,n_k$$

so that segment i_1 gets n_1 elements, segment i_2 gets n_2, and so on. The example above can be abbreviated to

```
DEFINE ELEMENTS
  SEG=1:4 EL=10,15,10,15 ; END
```

For this simple Processor allowing a command like this is probably overkill. It is implemented in that fashion only to illustrate the processing of variable length integer lists via CLVALI:

```
*
*    Define number of (equally spaced) boundary elements per segment
*
      subroutine    DEFINE_ELEMENTS
*
      implicit      none
      include       'SEGMENT.inc'
*
      character*4   key, CCLVAL
      integer       i, iseg, n, nseg
      integer       iseglist(MAXSEG), numelist(MAXSEG)
      integer       ICLTYP, ICLSEK
      real          FCLVAL
      logical       CMATCH
*
 1000   call      CLREAD  (' Element data> ',
     $     ' Enter SEG = i1 ... ik  EL = ne1, ... nek&&'//
     $          'Terminate with END')
*
```

```
      if (ICLTYP(1) .le. 0)            then
        print *, '*** Command must begin with keyword'
        go to 1000
      end if
      key =     CCLVAL(1)
      if (CMATCH (key, 'E^ND'))        then
         return
      else if (CMATCH (key, 'S^EG'))   then
        call  CLVALI (' ', -MAXSEG, iseglist, nseg)
        if (ICLSEK(0,'E^LEM') .eq. 0)  then
          print *, '*** Keyword ELEMENTS is missing'
          go to 1000
        end if
        call     CLVALI (' ', -MAXSEG, numelist, n)
        do 2500  i = 1,nseg
          iseg =  iseglist(i)
          if (iseg .le. 0 .or. iseg .gt. MAXSEG) then
            print *, '*** Segment number',iseg,' out of range'
          else
            numel(iseg) = max(numelist(i),1)
          end if
2500      continue
      else
        print *, '*** Illegal keyword ', key,' in element data'
      end if
      go to 1000
    end
```

If you can't follow the code, don't worry. It is more advanced than the typical input routine in DBEM2, so you can study it later.

Digression: Simplifying Commands

Why didn't we allow element data to be specified in the same commands that define the segment geometry? For example, we might have allowed commands such as

```
SEG = 13  BEG = -1.50,3.53  END = 14.81,6.22  ELEM = 5
```

The answer fits within another design principle:

> *Keep commands simple*

Paraphrasing Einstein: *A command should be as simple as possible, but no simpler.* Or Saint-Exupery: *you know that you have the perfect command when you can't remove anything.*

Simplicity is an admirable general principle, but for our case something more specific applies:

Don't mix persistent and volatile data in the same command

The terms "persistent" and "volatile" are used in a relative sense to denote degrees of "changeability" of the data. For example, segment data are more persistent than element data, since presumably you want to solve a problem whose geometry is dictated by external requirements; typically by engineering considerations. On the other hand, the number of elements per segment is a judgment decision: the program user attempts to get satisfactory accuracy (more elements, more accuracy) with reasonable cost (more elements, more computer time).

Frequently the number of elements is varied while keeping the segment data fixed; this is called a *convergence study.* So there are good reasons to separate the commands that define these two aspects.

Defining Boundary Conditions

Each segment may be given a different boundary condition (BC) that involves any of the following stress/displacement combinations:

BC Code	*Prescribed boundary values*
0	Shear stress σ_s and normal stress σ_n
1	Shear displacement u_s and normal displacement u_n
2	Shear displacement u_s and normal stress σ_n
3	Shear stress σ_s and normal displacement u_n

These values are *constant* along the segment, so they can be read on a segment-by-segment basis. The stress values are understood to be *resultants* over the segment.

REMARK B.1

The "BC codes" are related to those used by Crouch and Starfield. Using integer codes is far from the best way to implement readable software, but we shall follow their convention.

The BC data commands are introduced by a `DEFINE BOUNDARY_CONDITIONS` header command (which may be abbreviated to just `D B`), and have the form

$$\texttt{SEG} = i \quad \{\texttt{SS} = \sigma_s \mid \texttt{SD} = u_s\} \quad \{\texttt{NS} = \sigma_n \mid \texttt{ND} = u_n\}$$

terminated by an `END` command. Keyword `SS` means shear stress, `SD` shear displacement, and so on.

In the CLAMP metalanguage, this means that one may specify either σ_s or u_s, but not both simultaneously, and similarly for σ_n and u_n. The specifications are shown in braces, meaning that they may not be omitted.

If no BC is ever specified for segment i, that segment is assumed stress free (code 0 with $\sigma_s = \sigma_n = 0$). If only a normal value is prescribed, a zero shear stress is assumed, and so on.

The implementation of DEFINE_BOUNDARY follows.

```
*
*    Read boundary condition data for segments
*
      subroutine    DEFINE_BOUNDARY_CONDITIONS
C
      implicit      none
      include       'SEGMENT.inc'
*
      character*4   key, CCLVAL, word(2)
      integer       iseg, n, nw, iloc(2)
      integer       ICLVAL, ICLSEK, ICLTYP
      logical       CMATCH
*
 1000   call      CLREAD  (' Bound_cond data> ',
     $     ' Enter SEG=iseg {SS=sig_s | SD=u_s} {NS=sig_n | ND=u_n}'//
     $      '&&Terminate with END')
*
        if (ICLTYP(1) .le. 0)            then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))        then
          return
        else if (CMATCH (key, 'S^EG'))   then
          iseg =   ICLVAL(2)
          if (iseg .le. 0 .or. iseg .gt. MAXSEG)  then
            print *, '*** Segment number', iseg, ' is out of range'
            go to 1000
          end if
          call  CLOADK ('L', -2, word, iloc, nw)
          call  BCVALUES  (iseg, nw, word, iloc)
        else
          print *, '*** Illegal keyword ', key, ' in BC data'
        end if
        go to 1000
      end
```

This illustrates the use of the "load keyword" entry points of §8.2. These calls search for keywords such as SS and move them to the subroutine work area. This simplifies keyword legality tests such as "SS and SD cannot appear in the same command." To do these chores DEFINE_BOUNDARY calls subroutine BCVALUES:

```
*
*    Store boundary condition values in tables
*
      subroutine    BCVALUES
     $             (iseg, nw, word, iloc)
*
```

```
      implicit      none
      include       'SEGMENT.inc'
      character*(*) word(2)
      real          FCLVAL
      integer       iseg, nw, iloc(2)
      integer       code, i, isd, iloads, iloadn, ks, kd, kn
      logical       CMATCH
*
      ks =    0
      kn =    0
      kd =    0
      isd =   0
      iloadn =  0
      iloads =  0
*
      do 2000  i = 1,nw
        if (CMATCH (word(i), 'SS'))  then
          ks =   ks + 1
          iloads = iloc(i)
        else if (CMATCH (word(i), 'SD'))  then
          ks =   ks + 1
          kd =   kd + 1
          isd =  1
          iloads = iloc(i)
        else if (CMATCH (word(i), 'NS'))  then
          kn =   kn + 1
          iloadn = iloc(i)
        else if (CMATCH (word(i), 'ND'))  then
          kn =   kn + 1
          kd =   kd + 1
          iloadn = iloc(i)
        else
          print *, '*** Illegal BC keyword ', word(i),'  segment',iseg
          return
        end if
        if (kn .gt. 1 .or. ks .gt. 1)    then
          print *, '*** Illegal BC combination for segment', iseg
          return
        end if
 2000   continue
*
      if (iloadn .gt. 0)          bvn(iseg) =  FCLVAL(iloadn+1)
      if (iloads .gt. 0)          bvs(iseg) =  FCLVAL(iloads+1)
*
      if (kd .eq. 0)              then
        code =  1
      else if (kd .eq. 1)         then
        code =  3
        if (isd .eq. 0)           code = 4
      else
        code =  2
      end if
      kode(iseg) =  code-1
```

```
      return
      end
```

which embodies the logic for eventually storing the user-supplied values into appropriate spots in arrays BVS and BVN.

Defining Symmetry Conditions

If the problem exhibits symmetry conditions, commands to specify symmetry axes are introduced by the header command DEFINE SYMMETRY_CONDITIONS (which may be abbreviated to just D S) and have the form

$$\begin{aligned} \texttt{XSYM} &= x_{sym} \\ \texttt{YSYM} &= y_{sym} \end{aligned}$$

The XSYM command specifies that $x = x_{sym}$ is a line of symmetry parallel to the x axis. The YSYM command specifies that $y = y_{sym}$ is a line of symmetry parallel to the y axis. One or two specifications may be given. The Processor logic does not allow "skew" symmetry conditions.

The implementation of the DEFINE_SYMMETRY routine is straightforward:

```
*
*    Read symmetry condition data
*
      subroutine   DEFINE_SYMMETRY_CONDITIONS
*
      implicit      none
      include       'SYMMETRY.inc'
      character*4   key, CCLVAL, word(2)
      integer       ixsym, iysym, ICLTYP
      real          FCLVAL
      logical       CMATCH
*
      ixsym =  mod(ksym,2)
      iysym =  ksym/2
*
 1000   call      CLREAD  (' Symmetry data> ',
     $    ' Enter  XSYM=xsym  or  YSYM=ysym '//
     $     '&&Terminate with END')
        if (ICLTYP(1) .le. 0)            then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
*
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))       then
          ksym =  2*iysym + ixsym
          return
        else if (CMATCH (key, 'X^SYM')) then
```

```
      xsym =   FCLVAL(2)
      ixsym =  1
    else if (CMATCH (key, 'Y^SYM')) then
      ysym =   FCLVAL(2)
      iysym =  1
    else
      print *,  '*** Illegal keyword ', key,' in symmetry data'
    end if
    go to 1000
  end
```

REMARK B.2

Here KSYM is an integer "symmetry flag" related to that used in the original TWOBI program.

Defining Material Properties

Material properties are introduced by a DEFINE MATERIAL header command (which can be abbreviated to just D M). The commands have a simple form:

$$\texttt{EM} = E$$
$$\texttt{PR} = \nu$$

terminated by an END command. The E command specifies the elastic modulus and the PR command specifies Poisson's ratio. Since DBEM2 is restricted to elastic isotropic materials and does not consider thermal effects, these two material properties suffice.

The default values for E and ν set by CLEAR are 1.0 and 0.0, respectively.

The implementation of DEFINE MATERIALS is straightforward and does not involve any fancy new construct:

```
*
*     Read material property data
*
      subroutine    DEFINE_MATERIAL
*
      implicit      none
      include       'MATERIAL.inc'
      character*4   key, CCLVAL
      integer       ICLTYP
      real          FCLVAL
      logical       CMATCH
*
 1000   call       CLREAD  (' Material data> ',
     $              ' Enter EM=em  or  PR=pr&&'//
     $              'Terminate with END')
*
        if (ICLTYP(1) .le. 0)            then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))        then
          return
        else if (CMATCH (key, 'EM'))     then
          em =    FCLVAL(0)
        else if (CMATCH (key, 'P^R'))    then
          pr =    FCLVAL(0)
        else
          print *, '*** Illegal keyword ', key,' in material data'
        end if
        go to 1000
      end
```

Defining Prestress Data

If the initial stress state has nonzero components, prestress data have to be introduced by a DEFINE PRESTRESS header. The prestress-definition commands have a very simple form:

$$\begin{aligned} \texttt{SXXO} &= \sigma^0_{xx} \\ \texttt{SYYO} &= \sigma^0_{yy} \\ \texttt{SXYO} &= \sigma^0_{xy} \end{aligned}$$

As usual, these commands are terminated by an END command. Undefined prestress components are assumed zero.

The implementation of DEFINE_PRESTRESS is quite similar to that of DEFINE_MATERIAL :

```
*
*     Read prestress (initial field stresses) data
*
      subroutine    DEFINE_PRESTRESS
*
      implicit      none
      include       'PRESTRESS.inc'
      character*4   key, CCLVAL
      integer       ICLTYP
      real          FCLVAL
      logical       CMATCH
*
 1000   call       CLREAD  (' Prestress data> ',
     $              ' Enter SXXO=sxx0, SYYO=syy0 or SXYO=sxy0&&'//
     $              'Terminate with END')
*
        if (ICLTYP(1) .le. 0)             then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))         then
          return
        else if (CMATCH (key, 'SX^XO'))     then
          sxx0 =  FCLVAL(0)
        else if (CMATCH (key, 'SY^YO'))     then
          syy0 =  FCLVAL(0)
        else if (CMATCH (key, 'SX^YO'))     then
          sxy0 =  FCLVAL(0)
        else
          print *,'*** Illegal keyword ', key,' in prestress data'
        end if
        go to 1000
      end
```

Defining Output Field Locations

The last piece of input data are not related to the problem definition, but to the specification of the field points at which the program user would like to get computed results, *viz.*, displacements and stresses.

REMARK B.3

This set of information is characteristic of boundary element methods, in which all basic givens and unknowns are at the boundary. If you want information at field points not on the boundary, you have to ask for it and specify where.

For convenience the output locations are not specified point by point, but as equally spaced points on line segments. You specify the location of the first and last point on the line, and the number of points, if any, to be "collocated" between the first and last one.

The output field location specification commands are introduced by a **DEFINE FIELD_LOCATIONS** header command (which may be abbreviated to **D F**) and have a form reminiscent of the segment-definition commands:

$$\texttt{LINE} = i \quad \texttt{FIRST} = x_i^{first}, y_i^{first} \quad \texttt{LAST} = x_i^{last}, y_i^{last} \quad [\texttt{POINTS=}n_{int}]$$

Here n_{int} is the number of *intermediate* points to be inserted (equally spaced) between the first and last point. If this phrase is omitted, $n_{int} = 0$ is assumed so only the first and last points will be output points. If the first and last points coincide, output will be at only one point.

For example:

```
DEF OUT
  LINE=1 F=200.2 L=203.8 P=9
  LINE=2 F=3.8,0.2 L=0.2,3.8 P=9
  END
```

specifies two output lines running at 45° and 135°, respectively, and with 11 output points (first+last+9) in each.

Here is the implementation of the DEFINE_OUTPUT_LOCATIONS routine:

```
*
*     Read location of output field points
*
      subroutine   DEFINE_FIELD_LOCATIONS
*
      implicit      none
      include       'OUTPUT.inc'
      character*8   key, CCLVAL
      real          FCLVAL
      integer       ilin, n, mark, ICLVAL, ICLSEK, ICLTYP
      real          xy(2)
      logical       onepoint, CMATCH
*
 1000   call      CLREAD  (' Field location data> ',
     $   ' Enter LIN=ilin FIRST=xfirst,yfirst LAST=xlast,ylast'//
     $    '[P=ninter]&&Terminate with END')
*
        if (ICLTYP(1) .le. 0)            then
          print *, '*** Command must begin with keyword'
          go to 1000
        end if
        key =     CCLVAL(1)
        if (CMATCH (key, 'E^ND'))        then
          return
        else if (CMATCH (key, 'L^INE'))   then
          ilin =   ICLVAL(2)
          if (ilin .le. 0 .or. ilin .gt. MAXLIN)  then
            print *, '*** Field line number',ilin,' is out of range'
```

```
        go to 1000
      end if
      lindef(ilin) = 1
      nintop(ilin) = 0
      onepoint = .true.
      if (ICLSEK(3, 'F^IRST') .ne. 0)  then
        call   CLVALF (' ', 2, xy, n)
        if (n .ge. 1)   xfirst(ilin) = xy(1)
        if (n .ge. 2)   yfirst(ilin) = xy(2)
      end if
      if (ICLSEK(3, 'L^AST') .ne. 0)   then
        call   CLVALF (' ', 2, xy, n)
        if (n .ge. 1)   xlast(ilin) = xy(1)
        if (n .ge. 2)   ylast(ilin) = xy(2)
        onepoint = .false.
      end if
      if (onepoint)     then
        xlast(ilin) =  xfirst(ilin)
        ylast(ilin) =  yfirst(ilin)
      end if
      if (ICLSEK(3, 'P^OINTS') .ne. 0) then
        nintop(ilin) = max(ICLVAL(0),0)
      end if
    else
      print *, '*** Illegal keyword ', key,' in field loc data'
    end if
    go to 1000
  end
```

The input data section is complete.

§B.8 SOLVING THE PROBLEM

Having finished input data preparation, the three steps involved in solving the elastostatic problem are as follows.

Building the Boundary Element Model. The input data has defined the geometry of the problem in terms of segments. Segments are broken down into equally spaced boundary elements. The first step consists of building element-by-element data, and is carried out when you enter the command BUILD.

Assembling the Discrete Equations. This step generates a matrix **C** of "influence coefficients" and a vector **r** of "forcing functions." These arrays have dimensions equal to twice the total number of boundary elements. The construction of the elements of **C** and **r** follows the direct formulation of boundary-integral methods and is not explained here. This step is triggered by the command GENERATE and is carried out by subroutine GENERATE and subordinate routines.

Solving for the unknowns. The linear equation system $\mathbf{Cx} = \mathbf{r}$ is solved (by a Gauss elimination method) for vector **x**, which contains the boundary unknowns. This step is triggered by command SOLVE and is carried out by subroutine SOLVE and a subordinate routine.

Since we are not going to explain the theory behind these tasks, the BUILD, GENERATE and SOLVE subroutines are listed next without commentary.

```
*
*     Build detailed boundary element data
*
      subroutine  BUILD
*
      implicit      none
      include       'SEGMENT.inc'
      include       'ELEMENT.inc'
      include       'MATERIAL.inc'
      include       'PRESTRESS.inc'
      integer        iseg, k, ne, num
      real           xd, yd, side
*
      k =   0
      do 2000  iseg  = 1,MAXSEG
        if (segdef(iseg) .eq. 0)   go to 2000
        num =  numel(iseg)
        xd =   (xend(iseg)-xbeg(iseg))/num
        yd =   (yend(iseg)-ybeg(iseg))/num
        side = sqrt(xd**2+yd**2)
        if (side .eq. 0.0)         go to 2000
        do 1500  ne = 1,num
          k =   k + 1
          if (k .gt. MAXELM)     then
            print *, '*** Boundary element count exceeds ',MAXELM
            print *, '    Excess elements ignored'
```

```
            return
          end if
          xme(k) =  xbeg(iseg) + 0.5*(2.*ne-1)*xd
          yme(k) =  ybeg(iseg) + 0.5*(2.*ne-1)*yd
          hleng(k) = 0.5*side
          sinbet(k) = yd/side
          cosbet(k) = xd/side
          b(2*k-1) = bvs(iseg)
          b(2*k  ) = bvn(iseg)
          kod(k) =  kode(iseg)
 1500     continue
 2000   continue
        numbe =  k
        print '('' Discrete model building completed:'',
       $          I5,'' boundary elements''/)', numbe
        return
        end
*
*       Calculate influence coefficient matrix and RHS vector
*
        subroutine  GENERATE
*
        implicit       none
        include       'MATERIAL.inc'
        include       'ELEMENT.inc'
        include       'PRESTRESS.inc'
        include       'SYMMETRY.inc'
        integer        i, j
        real           sinbi, cosbi, sinbj, cosbj, ss0, sn0, g
        real           xi, xj, yi, yj, sj
        real           ass, asn, ans, ann, bss, bsn, bns, bnn
*
        g =    0.5*em/(1.+pr)
        do 3000  i = 1,numbe
          r(2*i-1) = 0.
          r(2*i  ) = 0.
          xi =  xme(i)
          yi =  yme(i)
          cosbi =  cosbet(i)
          sinbi =  sinbet(i)
          do 2500  j = 1,numbe
            ass =  0.0
            asn =  0.0
            ans =  0.0
            ann =  0.0
            bss =  0.0
            bsn =  0.0
            bns =  0.0
            bnn =  0.0
            xj =   xme(j)
            yj =   yme(j)
            cosbj =  cosbet(j)
            sinbj =  sinbet(j)
```

```
          sj =     hleng(j)
          ss0 = (syy0-sxx0)*sinbj*cosbj + sxy0*(cosbj**2-sinbj**2)
          sn0 = sxx0*sinbj**2 - 2.*sxy0*sinbj*cosbj + syy0*cosbj**2
          call   COEFF (xi, yi, xj, yj, sj,
     $                  1, em, pr, cosbi, sinbi, cosbj, sinbj,
     $                  ass, asn, ans, ann, bss, bsn, bns, bnn)
          if (ksym .eq. 1 .or. ksym .eq. 3)  then
            call   COEFF (xi, yi, 2.*xsym-xme(j), yj, sj,
     $                   -1, em, pr, cosbi, sinbi, cosbj, -sinbj,
     $                   ass, asn, ans, ann, bss, bsn, bns, bnn)
          end if
          if (ksym .eq. 2 .or. ksym .eq. 3)  then
            call   COEFF (xi, yi, xj, 2.*ysym-yme(j), sj,
     $                   -1, em, pr, cosbi, sinbi, -cosbj, sinbj,
     $                   ass, asn, ans, ann, bss, bsn, bns, bnn)
          end if
          if (ksym .eq. 3)                    then
            call   COEFF (xi, yi, 2.*xsym-xme(j), 2.*ysym-yme(j), sj,
     $                   1, em, pr, cosbi, sinbi, -cosbj, -sinbj,
     $                   ass, asn, ans, ann, bss, bsn, bns, bnn)
          end if
          call    SETUP  (i, j, kod(j), g, ss0, sn0,
     $                   ass, asn, ans, ann, bss, bsn, bns, bnn,
     $                   b, c, r, 2*numbe, MAXEQS)
 2500     continue
 3000   continue
        print *, 'Influence coefficient matrix & RHS vector generated'
        return
        end
*
*       Solve for unknown boundary values
*
        subroutine  SOLVE
*
        implicit      none
        include      'ELEMENT.inc'
        integer       ising
*
        call  GAUSSER (c, r, x, 2*numbe, MAXEQS, ising)
        if (ising .eq. 0)     then
          print *, 'Discrete equations solved'
        else
          print *, 'Singularity detected at BE equation',ising
        end if
        return
        end
```

Subroutine GENERATE calls COEFF (which is essentially the same as a TWOBI subroutine with the same name) and SETUP, which fills the entries of the influence coefficient matrix and right-hand-side vector:

```
*
```

```
*     Calculate source/receiver coefficients
*
      subroutine  COEFF
     $       (xi, yi, xj, yj, aj,
     $        msym, em, pr, cosbi, sinbi, cosb, sinb,
     $        ass, asn, ans, ann, bss, bsn, bns, bnn)
*
      implicit       none
      real           xi, yi, xj, yj, aj
      real           em, pr, cosbi, sinbi, cosb, sinb
      real           ass, asn, ans, ann, bss, bsn, bns, bnn
      real           pi, con, pr1, pr2, pr3
      integer        msym
      real           cma, cpa, cxb, cyb, cosg, sing
      real           r1s, r2s, fl1, fl2
      real           tb1, tb2, tb3, tb4, tb5
      real           asst, asnt, anst, annt
      real           bsst, bsnt, bnst, bnnt
*
      pi =   4.*atan2(1.,1.)
      con =  1.0/(4.*pi*(1.-pr))
      pr1 =  1.-2*pr
      pr2 =  2.*(1.-pr)
      pr3 =  3.-4.*pr
      cxb =  (xi-xj)*cosb + (yi-yj)*sinb
      cyb = -(xi-xj)*sinb + (yi-yj)*cosb
      cosg = cosbi*cosb + sinbi*sinb
      sing = sinbi*cosb - cosbi*sinb
*
      cma =  cxb - aj
      cpa =  cxb + aj
      r1s =  cma**2 + cyb**2
      r2s =  cpa**2 + cyb**2
      fl1 =   0.5*log(r1s)
      fl2 =   0.5*log(r2s)
      tb2 =  -con*(fl1-fl2)
      tb3 =  con*(atan2(cpa,cyb)-atan2(cma,cyb))
      tb1 =  -cyb*tb3 + con*(cma*fl1-cpa*fl2)
      tb4 =   con*(cyb/r1s-cyb/r2s)
      tb5 =   con*(cma/r1s-cpa/r2s)
*
      asst =   pr2*cosg*tb3 + pr1*sing*tb2 + cyb*(sing*tb4+cosg*tb5)
      asnt =  -pr1*cosg*tb2 + pr2*sing*tb3 + cyb*(cosg*tb4-sing*tb5)
      anst =  -pr2*sing*tb3 + pr1*cosg*tb2 + cyb*(cosg*tb4-sing*tb5)
      annt =   pr1*sing*tb2 + pr2*cosg*tb3 - cyb*(sing*tb4+cosg*tb5)
*
      bsst =   pr3*cosg*tb1 + cyb*(sing*tb2-cosg*tb3)
      bsnt =   pr3*sing*tb1 + cyb*(cosg*tb2+sing*tb3)
      bnst =  -pr3*sing*tb1 + cyb*(cosg*tb2+sing*tb3)
      bnnt =   pr3*cosg*tb1 - cyb*(sing*tb2-cosg*tb3)
*
      ass =    ass + msym*asst
      asn =    asn + asnt
```

```
      ans =    ans + msym*anst
      ann =    ann + annt
*
      bss =    bss + msym*bsst
      bsn =    bsn + bsnt
      bns =    bns + msym*bnst
      bnn =    bnn + bnnt
      return
      end
*
*     Set up influence coeff matrix and RHS of discrete system
*
      subroutine  SETUP
     $            (i, j, bckodj, g, ss0, sn0,
     $             ass, asn, ans, ann,
     $             bss, bsn, bns, bnn,
     $             b, c, r, n, nc)
*
      implicit      none
      integer       i, j, n, nc, bckodj
      real          ss0, sn0, g, bs, bn
      real          ass, asn, ans, ann, bss, bsn, bns, bnn
      real          b(*), c(nc,*), r(*)
*
      if (bckodj .eq. 0)          then
        c(2*i-1,2*j-1) =  ass
        c(2*i-1,2*j  ) =  asn
        c(2*i  ,2*j-1) =  ans
        c(2*i  ,2*j  ) =  ann
        bs =   0.5*(b(2*j-1)-ss0)/g
        bn =   0.5*(b(2*j  )-sn0)/g
        r(2*i-1) =  r(2*i-1) + bss*bs + bsn*bn
        r(2*i  ) =  r(2*i  ) + bns*bs + bnn*bn
      else if (bckodj .eq. 1)     then
        c(2*i-1,2*j-1) =  -bss
        c(2*i-1,2*j  ) =  -bsn
        c(2*i  ,2*j-1) =  -bns
        c(2*i  ,2*j  ) =  -bnn
        r(2*i-1) =  r(2*i-1) - ass*b(2*j-1) - asn*b(2*j)
        r(2*i  ) =  r(2*i  ) - ans*b(2*j-1) - ann*b(2*j)
      else if (bckodj .eq. 2)     then
        c(2*i-1,2*j-1) =  -bss
        c(2*i-1,2*j  ) =   asn
        c(2*i  ,2*j-1) =  -bns
        c(2*i  ,2*j  ) =  ann
        bn =   0.5*(b(2*j  )-sn0)/g
        r(2*i-1) =  r(2*i-1) - ass*b(2*j-1) + bsn*bn
        r(2*i  ) =  r(2*i  ) - ans*b(2*j-1) + bnn*bn
      else
        c(2*i-1,2*j-1) =  ass
        c(2*i-1,2*j  ) =  -bsn
        c(2*i  ,2*j-1) =  ans
        c(2*i  ,2*j  ) =  -bnn
```

```
      bs =   0.5*(b(2*j-1)-ss0)/g
      r(2*i-1) =  r(2*i-1) + bss*bs - asn*b(2*j)
      r(2*i  ) =  r(2*i  ) + bns*bs - ann*b(2*j)
    end if
    return
    end
```

SOLVE calls GAUSSER, which is a naïve implementation of unsymmetric Gauss elimination without pivoting:

```
*
*     Solve algebraic equation system A x = b by Gauss elimination
*
      subroutine  GAUSSER
     $           (a, b, x, n, na, ising)
*
      implicit      none
      integer       n, na, ising
      real          a(na,*), b(*), x(*), c, sum
      integer       i, j, k
*
      ising = 0
      do 2000  j = 1,n-1
        if (a(j,j) .eq. 0.0)   then
          ising =  j
          return
        end if
        do 1500  k = j+1,n
          c =   a(k,j)/a(j,j)
          do 1400  i = j,n
            a(k,i) = a(k,i) - c*a(j,i)
 1400       continue
          b(k) = b(k) - c*b(j)
 1500     continue
 2000   continue
*
      x(n) =  b(n)/a(n,n)
      do 3000  j = n-1,1,-1
        sum = 0.0
        do 2500  i = j+1,n
          sum =  sum + a(j,i)*x(i)
 2500     continue
        x(j) = (b(j)-sum)/a(j,j)
 3000   continue
      return
      end
```

(The only redeeming quality about GAUSSER is that the code is quite short; in fact, it's about the shortest possible implementation of a linear equation solver.)

§B.9 PRINTING DATA

One area in which the interactive operation excels is data display. If you are using an interactive Processor for a engineering design task, you can selectively trim the otherwise voluminous output to the important essentials. Conversely, if you are debugging a new or modified implementation, you may want more output than is normally required; for example, printing the influence coefficient matrix.

What applies for printed output applies with equal force to graphic output. We are not going to illustrate graphic displays here, however, since the details depend strongly on the output device and the plotting software you are using.

The PRINT command is similar to the DEFINE command in that it takes a second keyword that specifies what is to be printed:

SEGMENTS	Prints segment geometry data and number of elements per segment.
BOUNDARY_CONDITIONS	Prints boundary condition (BC) code and prescribed boundary values for each segment.
SYMMETRY_CONDITIONS	Prints symmetry conditions if any are in effect.
MATERIAL	Prints material property data.
PRESTRESS	Prints prestress data.
FIELD_LOCATIONS	Prints information about output-location lines if any are defined.
ELEMENTS	Prints detailed boundary-element data produced by subroutine BUILD (this is primarily for debugging).
COEFFICIENTS	Prints the matrix C of influence coefficients assembled by GENERATE (this is primarily for debugging).
RHS	Prints the right-hand side (forcing) vector r assembled by GENERATE (this is primarily for debugging).
SOLUTION	Prints the solution vector x calculated by SOLVE (this is primarily for debugging).
RESULTS	Print stresses and displacements at boundary-element midpoints or at output field locations, depending on a command qualifier.

The PRINT command is processed by subroutine PRINT, which has a "case" structure similar to that of subroutine DEFINE:

```
*
*     Interpret PRINT command
*
      subroutine    PRINT
*
      implicit      none
      character     key*8, CCLVAL*8
      integer       ICLTYP
      logical       CMATCH
*
      if (ICLTYP(2) .le. 0)  then
        call   CLREAD (' PRINT what? ',
     $    ' BOUNDARY, ELEMENTS, COEFFICIENTS,'//
     $     'FIELD, MATERIAL, PRESTRESS&&'//
     $     'RESULTS, RHS, SOLUTION, SYMMETRY')
        key =  CCLVAL(1)
      else
        key =  CCLVAL(2)
      end if
      if (CMATCH (key, 'B^OUNDARY'))    then
        call  PRINT_BOUNDARY_CONDITIONS
      else if (CMATCH (key, 'C^OEFFICIENTS') .or.
     $         CMATCH (key, 'I^NFLUENCE')) then
        call  PRINT_INFLUENCE_COEFFICIENTS
      else if (CMATCH (key, 'E^LEMENTS')) then
        call  PRINT_ELEMENTS
      else if (CMATCH (key, 'F^IELD'))   then
        call  PRINT_FIELD_LOCATIONS
      else if (CMATCH (key, 'M^ATERIAL')) then
        call  PRINT_MATERIAL
      else if (CMATCH (key, 'P^RESTRESS')) then
        call  PRINT_PRESTRESS
      else if (CMATCH (key, 'RE^SULTS'))  then
        call  PRINT_RESULTS
      else if (CMATCH (key, 'RHS'))       then
        call  PRINT_RHS_VECTOR
      else if (CMATCH (key, 'SE^GMENT'))  then
        call  PRINT_SEGMENTS
      else if (CMATCH (key, 'S^OLUTION')) then
        call  PRINT_SOLUTION_VECTOR
      else if (CMATCH (key, 'SY^MMETRY')) then
        call  PRINT_SYMMETRY_CONDITIONS
      else
        print *,'*** Illegal or ambiguous keyword ',key,' after PRINT'
      end if
      return
      end
```

Subroutine PRINT provides our first (and only) example of an implementation that *prompts for missing data.* If you type only the keyword PRINT followed by a carriage return, you will see the prompt

Print what?

on the screen, and you are supposed to type the next keyword, *e.g.*, SEGMENTS that you forgot. (Notice that this friendly technique was not used for the DEFINE command explained in §B.7; instead subroutine DEFINE complains about missing keywords after DEFINE.)

Next we examine the subordinate routines.

Printing Input Data

The implementation of the subroutines that print segment, boundary condition, symmetry, material, prestress, and field-location data are straightforward and so are simply listed next as a group:

```
*
*     Print segment data
*
      subroutine   PRINT_SEGMENTS
*
      implicit      none
      include       'SEGMENT.inc'
      integer       i, k
*
      k =    0
      do 2000  i = 1,MAXSEG
        if (segdef(i) .gt. 0)      then
          if (k .eq. 0)            then
            print '(/A/A6,A9,4A12)',
     $            ' Boundary Segment Data',
     $            'Segm', 'Elements', 'Xbeg', 'Ybeg', 'Xend', 'Yend'
          end if
          k =    k + 1
          print '(I6,I9,3X,4G12.4)',
     $           i, numel(i), xbeg(i), ybeg(i), xend(i), yend(i)
        end if
 2000   continue
      if (k .eq. 0)          then
        print *, 'Segment tables are empty'
      end if
      print *, ' '
      return
      end
*
*     Print boundary data in response to a PRINT BOUNDARY command
*
      subroutine   PRINT_BOUNDARY_CONDITIONS
*
      implicit      none
      include       'SEGMENT.inc'
*
      integer       i, k
      character*9   given(0:3)
```

```
      data    given /'SS and NS', 'SD and ND', 'SD and NS', 'SS and ND'/
*
      k =    0
      do 2000  i = 1,MAXSEG
        if (segdef(i) .gt. 0)     then
          if (k .eq. 0)           then
            print '(/A/A6,A11,2A12)',
     $            ' Boundary Conditions Data', 'Segm',
     $            'Given',  'Shear', 'Normal'
          end if
          k =    k + 1
          print '(I5,1X,A11,3X,1P2G12.3)',
     $          i, given(kode(i)), bvs(i), bvn(i)
        end if
 2000   continue
*
      if (k .eq. 0)          then
        print *, 'Boundary tables are empty'
      end if
      print *, ' '
      return
      end
*
*     Print symmetry data
*
      subroutine   PRINT_SYMMETRY_CONDITIONS
*
      implicit      none
      include       'SYMMETRY.inc'
*
      print '(/A)', ' Symmetry Data'
      if (ksym .eq. 3)            then
        print *, 'Symmetry about axis X=',xsym
        print *, '            and axis Y=',ysym
      else if (ksym .eq. 1)       then
        print *, 'Symmetry about axis X=',xsym
      else if (ksym .eq. 2)       then
        print *, 'Symmetry about axis Y=',ysym
      else
        print *, 'No symmetry conditions'
      end if
      print *, ' '
      return
      end
*
*     Print material property data
*
      subroutine   PRINT_MATERIAL
*
      implicit      none
      include       'MATERIAL.inc'
*
      print '(/A)', ' Material Property Data'
```

```
      print '('' Elastic modulus:'',1PE12.3)', em
      print '('' Poisson''''s ratio:'',F12.3)', pr
      print *, ' '
      return
      end
*
*     Print field location data
*
      subroutine   PRINT_FIELD_LOCATIONS
*
      implicit      none
      include      'OUTPUT.inc'
      integer       i, k
*
      k =    0
      do 2000  i = 1,MAXLIN
        if (lindef(i) .gt. 0)      then
          if (k .eq. 0)            then
            print '(/A/A6,A9,A9,3A12)',
     $            ' Field Location Data',
     $            'Line', 'Int.Pts', 'x-first', 'y-first',
     $            'x-last', 'y-last'
          end if
          k =    k + 1
          print '(I6,I9,4G12.4)',
     $          i, nintop(i), xfirst(i), yfirst(i), xlast(i), ylast(i)
        end if
 2000   continue
*
      if (k .eq. 0)          then
        print *, 'FIELD Location Tables are empty'
      end if
      print *, ' '
      return
      end
```

Debug-Oriented Print Commands

The PRINT ELEMENTS, PRINT COEFFICIENTS, PRINT RHS and PRINT SOLUTION are detailed print commands primarily useful in debug situations. They are implemented in the following subroutines:

```
*
*     Print detailed boundary element data
*
      subroutine  PRINT_ELEMENTS
*
      implicit      none
*     include       'SEGMENT.inc'
      include       'ELEMENT.inc'
      integer        m
*
```

```
      if (numbe .le. 0)           then
        print *, 'Boundary element table empty'
        return
      end if
      print '(/A/A5,A8,2A11,A12,A8,A9,A12)',
     $      ' Boundary Element Data',
     $      'Elem', 'Xmid', 'Ymid', 'Length',
     $      'Orient', 'BCode', 'Shear', 'Normal'
      do 2000  m = 1,numbe
        print '(I5,1P3G11.3,OPF10.2,I6,1P2G12.3)',
     $         m,xme(m),yme(m),2.*hleng(m),
     $         (180./3.14159265)*atan2(sinbet(m),cosbet(m)),
     $         kod(m), b(2*m-1),b(2*m)
 2000   continue
      print *, ' '
      return
      end
*
*     Print influence coefficient matrix
*
      subroutine  PRINT_INFLUENCE_COEFFICIENTS
*
      implicit      none
*     include       'SEGMENT.inc'
      include       'ELEMENT.inc'
*
      print '(/A)', ' Influence Coefficient Matrix'
      call   PRINT_REAL_MATRIX (c, MAXEQS, 2*numbe, 2*numbe)
      print *, ' '
      return
      end
*
*      Print right hand side vector
*
      subroutine PRINT_RHS_VECTOR
      implicit      none
*     include       'SEGMENT.inc'
      include       'ELEMENT.inc'
*
      print '(/A)', ' Right Hand Side (Forcing) Vector'
      call   PRINT_REAL_MATRIX (r, 1, 1, 2*numbe)
      print *, ' '
      return
      end
*
*      Print right hand side vector
*
      subroutine PRINT_SOLUTION_VECTOR
      implicit      none
*     include       'SEGMENT.inc'
      include       'ELEMENT.inc'
*
      print '(/A)', ' Solution Vector'
```

```
      call   PRINT_REAL_MATRIX (x, 1, 1, 2*numbe)
      print *, ' '
      return
      end
```

The previous three subroutines call PRINT_REAL_MATRIX, which is a "no frills" array printer:

```
*
*      Print real matrix (or vector) in 6-column template
*
      subroutine PRINT_REAL_MATRIX
     $      (a, na, m, n)
      integer   na, m, n, i, j, jref
      real      a(na,*)
      do 4000  jref = 0,n-1,6
        print '(1X,6I12)', (j,j=jref+1,min(jref+6,n))
        do 3000  i = 1,m
          print '(I4,1P6E12.4)', i,(a(i,j),j=jref+1,min(jref+6,n))
3000      continue
4000    continue
       return
       end
```

Printing Results

The PRINT RESULTS command without a qualifier lists stresses and displacements computed at boundary element midpoints. If qualifier FIELD appears, the command refers to the field points previously defined. This switch is implemented in subroutine PRINT_RESULTS:

```
*
*     Process PRINT RESULTS command
*
      subroutine   PRINT_RESULTS
*
      implicit      none
      integer       ICLSEQ
*
      if (ICLSEQ(3,'F^IELD') .eq. 0)   then
        call  PRINT_BOUNDARY_RESULTS
      else
        call  PRINT_FIELD_RESULTS
      end if
      return
      end
```

The code above provides an example of the use of ICLSEQ to test for the existence of a specific qualifier, in this case FIELD.

REMARK B.4

Admittedly the use of a qualifier here is somewhat contrived, for using the command form **PRINT RESULTS FIELD** would be perfectly acceptable. The qualifier form is merely selected only to illustrate the use of ICLSEQ. Generally speaking, the use of qualifiers is appropriate only for more complex Processors than DBEM2.

Printing Boundary Results

This is done by subroutine PRINT_BOUNDARY_RESULTS, the implementation of which is straightforward:

```
*
*     Print stresses and displacement @ boundary element midpoints
*
      subroutine  PRINT_BOUNDARY_RESULTS
*
      implicit      none
      include       'SEGMENT.inc'
      include       'ELEMENT.inc'
      include       'MATERIAL.inc'
      include       'PRESTRESS.inc'
      integer        k
      real           g, ss0, sn0, sinbi, cosbi
      real           us, un, ux, uy, sign, sigs
```

```
*
      print '(/A)', ' Displacements and Stresses at'//
     $              ' Boundary Element Midpoints'
      print '(A5,A9,5A11)', 'Elem', 'u_s', 'u_n', 'u_x', 'u_y',
     $                     'sig_s', 'sig_n'
      g =  0.5*em/(1.+pr)
*
      do 2000  k = 1,numbe
        us =    x(2*k-1)
        un =    x(2*k  )
        sigs =  b(2*k-1)
        sign =  b(2*k  )
        if (kod(k) .eq. 1)       then
          un =  b(2*k-1)
          us =  b(2*k  )
          sigs = x(2*k-1)
          sign = x(2*k)
        else if (kod(k) .eq. 2)  then
          us =  b(2*k-1)
          sigs =  x(2*k-1)
        else if (kod(k) .eq. 3)  then
          un =  b(2*k  )
          sign = b(2*k  )
        end if
        sinbi = sinbet(k)
        cosbi = cosbet(k)
        ux = us*cosbi - un*sinbi
        uy = us*sinbi + un*cosbi
        print '(I5,1P6G11.3)', k, us,un,ux,uy,sigs,sign
 2000   continue
      print *, ' '
      return
      end
```

Printing Field Results

Showing displacements and stresses at field points is complicated by the fact that, unlike finite element programs, such values are not readily available but must be calculated as part of the display procedure. This will become evident as one shows the coding of subroutine PRINT_FIELD_RESULTS:

```
*
*     Print stresses and displacements @ specified field points
*
      subroutine  PRINT_FIELD_RESULTS
*
      implicit       none
      include       'OUTPUT.inc'
      integer        m, p, points
      real           xp, yp, ux, uy, sigxx, sigyy, sigxy, f
      logical        skip
*
```

```
      print '(/A)', ' Displacements and Stresses at'//
     $             ' Specified Field Points'
*
      do 3000  m = 1,MAXLIN
        if (lindef(m) .eq. 0)    go to 3000
        print '(A5,2A10,A8,4A11)', ' Lin', 'x', 'y', 'u_x', 'u_y',
     $                      'sig_xx', 'sig_yy', 'sig_xy'
        points = nintop(m) + 2
        if (xfirst(m) .eq. xlast(m) .and.
     $      yfirst(m) .eq. ylast(m))  points = 1
        f =  0.0
        do 2000  p = 1,points
          if (points .gt. 1)  f =  real(p-1)/(points-1)
          xp = xfirst(m)*(1.0-f) + xlast(m)*f
          yp = yfirst(m)*(1.0-f) + ylast(m)*f
          call  FIELDP (xp, yp, ux, uy, sigxx, sigyy, sigxy, skip)
          if (skip)    then
            print '(I5,2F10.3,6X,A)', m, xp,yp,
     $            'Point is too close to boundary'
          else
            print '(I5,2F10.3,1P5G11.3)', m, xp,yp, ux,uy,
     $                                   sigxx,sigyy,sigxy
          end if
 2000     continue
        print *, ' '
 3000   continue
      return
      end
```

Subroutine FIELDP receives the location XP,YP of the field point and returns the displacement components u_x and u_y, and the stress components σ_{xx}, σ_{yy} and σ_{xy}:

```
*
*     Compute stresses and displacements at field point
*
      subroutine  FIELDP
     $           (xp, yp,
     $            ux, uy, sigxx, sigyy, sigxy, skip)
*
      implicit      none
*     include       'SEGMENT.inc'
      include       'ELEMENT.inc'
      include       'MATERIAL.inc'
      include       'SYMMETRY.inc'
      include       'PRESTRESS.inc'
      real           xp, yp, us, un, ux, uy, sigxx, sigyy, sigxy
      logical        skip
      real           uxus, uxun, uxss, uxsn
      real           uyus, uyun, uyss, uysn
      real           sxxus, sxxun, sxxss, sxxsn
      real           syyus, syyun, syyss, syysn
      real           sxyus, sxyun, sxyss, sxysn
      real           xj, yj, sj, cosbj, sinbj
```

```
      real           usj, unj, ssj, snj, ssg, sng
      real           g, ss0, sn0
      integer        j
*
      skip =  .false.
      ux =  0.0
      uy =  0.0
      sigxx =  sxx0
      sigyy =  syy0
      sigxy =  sxy0
      g =     0.5*em/(1.+pr)
*
      do 2000  j = 1,numbe
        uxus =  0.0
        uxun =  0.0
        uxss =  0.0
        uxsn =  0.0
        uyun =  0.0
        uyun =  0.0
        uyss =  0.0
        uysn =  0.0
        sxxus = 0.0
        sxxun = 0.0
        sxxss = 0.0
        sxxsn = 0.0
        syyus = 0.0
        syyun = 0.0
        syyss = 0.0
        syysn = 0.0
        sxyus = 0.0
        sxyun = 0.0
        sxyss = 0.0
        sxysn = 0.0
        xj =  xme(j)
        yj =  yme(j)
        sj =  hleng(j)
        if ((xp-xj)**2+(yp-yj)**2 .le. (   sj)**2)  then
          skip =  .true.
          return
        end if
        cosbj = cosbet(j)
        sinbj = sinbet(j)
        ss0 = (syy0-sxx0)*sinbj*cosbj + sxy0*(cosbj**2-sinbj**2)
        sn0 = sxx0*sinbj**2 - 2.*sxy0*sinbj*cosbj + syy0*cosbj**2
        call   SOMIGLIANA (xp, yp, xj, yj, sj,
     $                1, em, pr, cosbj, sinbj,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
        if (ksym .eq. 1 .or. ksym .eq. 3)  then
          call   SOMIGLIANA (xp, yp, 2.*xsym-xme(j), yj, sj,
```

```
     $                -1, em, pr, cosbj, -sinbj,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
        end if
        if (ksym .eq. 2 .or. ksym .eq. 3)  then
          call   SOMIGLIANA (xp, yp, xj, 2.*ysym-yme(j), sj,
     $                -1, em, pr, -cosbj, sinbj,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
        end if
        if (ksym .eq. 3)                    then
          call   SOMIGLIANA (xp, yp, 2.*xsym-xme(j), 2.*ysym-yme(j), sj,
     $                1, em, pr, -cosbj, -sinbj,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
        end if
        usj = x(2*j-1)
        unj = x(2*j  )
        ssj = b(2*j-1) - ss0
        snj = b(2*j  ) - sn0
        if (kod(j) .eq. 1)   then
          usj =  b(2*j-1)
          unj =  b(2*j  )
          ssj =  x(2*j-1)
          snj =  x(2*j  )
        else if (kod(j) .eq. 2)  then
          usj =  b(2*j-1)
          ssj =  x(2*j  )
        else if (kod(j) .eq. 3)  then
          unj =  b(2*j  )
          snj =  x(2*j  )
        end if
        ssg = 0.5*ssj/g
        sng = 0.5*snj/g
        ux =  ux + uxus*usj + uxun*unj + uxss*ssg + uxsn*sng
        uy =  uy + uyus*usj + uyun*unj + uyss*ssg + uysn*sng
        usj = 2.*g*usj
        unj = 2.*g*unj
        sigxx = sigxx + sxxus*usj + sxxun*unj + sxxss*ssj + sxxsn*snj
        sigyy = sigyy + syyus*usj + syyun*unj + syyss*ssj + syysn*snj
        sigxy = sigxy + sxyus*usj + sxyun*unj + sxyss*ssj + sxysn*snj
2000    continue
      return
      end
```

Finally, FIELDP calls subroutine SOMIGLIANA to evaluate the important boundary-on-field-point influence coefficients:

```
*
*     Calculate field influence coefficients from Somigliana's formula
*
      subroutine  SOMIGLIANA
     $    (x, y, xj, yj, aj, msym, em, pr, cosb, sinb,
     $                uxus, uxun, uxss, uxsn,
     $                uyus, uyun, uyss, uysn,
     $                sxxus, sxxun, sxxss, sxxsn,
     $                syyus, syyun, syyss, syysn,
     $                sxyus, sxyun, sxyss, sxysn)
*
      implicit      none
      real           x, y, xj, yj, aj, em, pr, cosb, sinb
      real           uxus, uxun, uxss, uxsn
      real           uyus, uyun, uyss, uysn
      real           sxxus, sxxun, sxxss, sxxsn
      real           syyus, syyun, syyss, syysn
      real           sxyus, sxyun, sxyss, sxysn
      integer        msym
      real           pi, con, pr1, pr2, pr3
      real           cxb, cyb, cosg, sing, cpa, cma
      real           r1s, r2s, fl1, fl2
      real           tb1, tb2, tb3, tb4, tb5, tb6, tb7
      real           uxust, uxunt, uxsst, uxsnt
      real           uyust, uyunt, uysst, uysnt
      real           sxxust, sxxunt, sxxsst, sxxsnt
      real           syyust, syyunt, syysst, syysnt
      real           sxyust, sxyunt, sxysst, sxysnt
      real           cosb2, sinb2, cos2b, sin2b
*
      pi =   4.*atan2(1.,1.)
      con =  1.0/(4.*pi*(1.-pr))
      pr1 =  1.-2*pr
      pr2 =  2.*(1.-pr)
      pr3 =  3.-4.*pr
*
      cxb =  (x-xj)*cosb + (y-yj)*sinb
      cyb = -(x-xj)*sinb + (y-yj)*cosb
*
      cma =  cxb - aj
      cpa =  cxb + aj
      r1s =  cma**2 + cyb**2
      r2s =  cpa**2 + cyb**2
      fl1 =   0.5*log(r1s)
      fl2 =   0.5*log(r2s)
      tb2 =  -con*(fl1-fl2)
      tb3 =  con*(atan2(cpa,cyb)-atan2(cma,cyb))
      tb1 =  -cyb*tb3 + con*(cma*fl1-cpa*fl2)
      tb4 =   con*(cyb/r1s-cyb/r2s)
      tb5 =   con*(cma/r1s-cpa/r2s)
```

```
      tb6 =  con*((cma**2-cyb**2)/r1s**2-(cpa**2-cyb**2)/r2s**2)
      tb7 =  -con*2.*cyb*(cma/r1s**2-cpa/r2s**2)
*
      uxust =  pr1*sinb*tb2 - pr2*cosb*tb3 + cyb*(sinb*tb4-cosb*tb5)
      uxunt =  pr1*cosb*tb2 + pr2*sinb*tb3 - cyb*(cosb*tb4+sinb*tb5)
      uxsst =  pr3*cosb*tb1 - cyb*(sinb*tb2+cosb*tb3)
      uxsnt = -pr3*sinb*tb1 + cyb*(cosb*tb2-sinb*tb3)
      uyust = -pr1*cosb*tb2 - pr2*sinb*tb3 - cyb*(cosb*tb4+sinb*tb5)
      uyunt =  pr1*sinb*tb2 - pr2*cosb*tb3 - cyb*(sinb*tb4-cosb*tb5)
      uysst =  pr3*sinb*tb1 + cyb*(cosb*tb2-sinb*tb3)
      uysnt =  pr3*cosb*tb1 + cyb*(sinb*tb2+cosb*tb3)
*
      cosb2 = cosb*cosb
      sinb2 = sinb*sinb
      cos2b = cosb2-sinb2
      sin2b = 2.*sinb*cosb
*
      sxxust = 2.*cosb2*tb4 + sin2b*tb5 - cyb*(cos2b*tb6-sin2b*tb7)
      syyust = 2.*sinb2*tb4 - sin2b*tb5 + cyb*(cos2b*tb6-sin2b*tb7)
      sxyust = sin2b*tb4 - cos2b*tb5 - cyb*(sin2b*tb6+cos2b*tb7)
      sxxunt = -tb5 - cyb*(sin2b*tb6+cos2b*tb7)
      syyunt = -tb5 + cyb*(sin2b*tb6+cos2b*tb7)
      sxyunt =  cyb*(cos2b*tb6-sin2b*tb7)
      sxxsst = -tb2 - pr2*(cos2b*tb2-sin2b*tb3)
     $         + cyb*(cos2b*tb4+sin2b*tb5)
      syysst = -tb2 - pr2*(cos2b*tb2-sin2b*tb3)
     $         - cyb*(cos2b*tb4+sin2b*tb5)
      sxysst = - pr2*(sin2b*tb2+cos2b*tb3)
     $         + cyb*(sin2b*tb4-cos2b*tb5)
      sxxsnt = -tb3 + pr1*(sin2b*tb2+cos2b*tb3)
     $         + cyb*(sin2b*tb4-cos2b*tb5)
      syysnt = -tb3 - pr1*(sin2b*tb2+cos2b*tb3)
     $         - cyb*(sin2b*tb4-cos2b*tb5)
      sxysnt = - pr1*(cos2b*tb2-sin2b*tb3)
     $         - cyb*(cos2b*tb4+sin2b*tb5)
*
      uxus =    uxus + msym*uxust
      uxun =    uxun + uxunt
      uxss =    uxss + msym*uxsst
      uxsn =    uxsn + uxsnt
      uyus =    uyus + msym*uyust
      uyun =    uyun + uyunt
      uyss =    uyss + msym*uysst
      uysn =    uysn + uysnt
*
      sxxus =    sxxus + msym*sxxust
      sxxun =    sxxun + sxxunt
      sxxss =    sxxss + msym*sxxsst
      sxxsn =    sxxsn + sxxsnt
      syyus =    syyus + msym*syyust
      syyun =    syyun + syyunt
      syyss =    syyss + msym*syysst
      syysn =    syysn + syysnt
```

```
      sxyus =     sxyus + msym*sxyust
      sxyun =     sxyun + sxyunt
      sxyss =     sxyss + msym*sxysst
      sxysn =     sxysn + sxysnt
*
      return
      end
```

The DBEM2 Processor is complete.

§B.10 DBEM2 MODULE STRUCTURE

After all the coding details given in §B.5 through §B.9 it is perhaps refreshing to get an overall picture of the structure of DBEM2. A *hierarchical diagram* of the module structure can provide such a picture:

```
DBEM2
  DOCOMMAND
    BUILD
    CLEAR
    DEFINE
      DEFINE_BOUNDARY_CONDITIONS
        BCVALUES
      DEFINE_ELEMENTS
      DEFINE_MATERIAL
      DEFINE_FIELD_LOCATIONS
      DEFINE_PRESTRESS
      DEFINE_SEGMENTS
      DEFINE_SYMMETRY
    GENERATE
      COEFF
      SETUP
    PRINT
      PRINT_BOUNDARY_CONDITIONS
      PRINT_BOUNDARY_RESULTS
      PRINT_COEFFICIENTS
        PRINT_REAL_MATRIX
      PRINT_ELEMENTS
      PRINT_FIELD_RESULTS
        FIELDP
          SOMIGLIANA
      PRINT_MATERIAL
      PRINT_PRESTRESS
      PRINT_RHS
        PRINT_REAL MATRIX
      PRINT_SEGMENTS
      PRINT_SOLUTION
        PRINT_REAL_MATRIX
      PRINT_SYMMETRY
    SOLVE
      GAUSSER
    STOP
```

This diagram of course excludes the NICE utilities such as the CLIP system. With this omission noted, the deepest module level is five. This is a feature symptomatic of a fairly

simple Processor. (Actual production Processors in the NICE system reach module levels of order 15–20.)

§B.11 AN EXAMPLE PROBLEM

It is convenient to test DBEM2 on the same example problem used in Crouch and Starfield (ref. B-1). The problem concerns a unit-radius circular hole in an infinite body under uniaxial tension at infinity. The boundary element discretization for one-quarter of the hole is shown in Figure B.2.

Figure B.2. Circular hole in an infinite body:
(a) problem specifications, (b) boundary element model

Both $x = 0$ and $y = 0$ are symmetry lines. The boundary contour is approximated by six straight-line segments, each of which consists of one boundary element. Two field point lines are chosen along portions of the x and y axes as shown in Figure B.2(a).

The input for this problem is prepared (with the text editor) in the form of a script command file:

```
clear
def segments
  seg=1 b=1,0          e=.9659,.2588
  seg=2 b=.9659,.2588 e=.8660,.5000
  seg=3 b=.8660,.5000 e=.7071,.7071
  seg=4 b=.7071,.7071 e=.5000,.8660
  seg=5 b=.5000,.8660 e=.2588,.9659
  seg=6 b=.2588,.9659 e=0,1
  end
def material
  em=7.E4 ; pr=0.2 ; end
def symmetry
  xsym=0 ; ysym = 0 ; end
def prestress
  sxx0=100 ; end
def field
  line=1 f=1,0 l=6,0 p=9
  line=2 f=0,1 l=0,6 p=9
  end
pri seg ; pri mat ; pri bou ; pri symm ; pri pres ; pri field
build ; gen ; sol
pri res ; pri res/field
stop
```

Note that there is no need for DEFINE ELEMENT input data because each segment contains only one boundary element, which is the default assumption.

Upon starting the DBEM2 processor, this file is inserted in the command stream through an ADD directive. For example, under VAX/VMS:

```
$ RUN DBEM2
  DBEM2> *ADD CIRCHOLE.ADD
```

where CIRCHOLE.ADD is the assumed name of the input file. The printed results should then be compared with those given in Appendix C of Crouch and Starfield (ref. B-1).

References:

B-1 Crouch S. L. and Starfield, A. M., *Boundary Element Methods in Solid Mechanics: with Applications in Rock Mechanics and Geological Engineering*. G. Allen and Unwin, London, 1983.

THIS PAGE LEFT BLANK INTENTIONALLY.

C
Help Files

§C.1 BACKGROUND

CLIP provides keyword-driven online help display services through the **HFILE** and **HELP** directives documented in Volume II. Online help text does not reside in the Processors themselves, but on separated card-image files known as *help files*. To take advantage of the display services, the structure of help files must conform to the technical specification presented in this Appendix.

The organization of this Appendix is as follows. §C.2 gives an overview of the NICE online help philosophy. §C.3 and §C.4 go to the heart of the matter and cover technical details that should be mastered before you attempt to write help files for your Processors. The exposition relies heavily on an example file prepared for the DBEM2 (Two-Dimensional Directly-formulated Boundary Element Method) Processor documented in Appendix B. §C.5 illustrates the use of messages to implement HELP commands for inexperienced users. Finally, §C.6 shows the complete help file for DBEM2.

The help file structure described here was originally devised by Charles Perry in 1980 for the NICE demonstration Processors MUP and SNAP.

§C.2 HELP FILE ORGANIZATION OVERVIEW

The structure of a NICE help file closely mimics that of the VAX/VMS online system help file, which served as inspiration for the original design. The NICE implementation, however, is not restricted to specific computers.

A NICE help file is a *tree of information* that maps into a *sequential file organization* readable with standard formatted FORTRAN I/O. The mapping is controlled by *sentinel characters* stored in the first column of each record (card image, line) of the file. These characters do not show up on display. The arrangement of the information is such that the file reader *never has to backspace* over previously read lines while "traversing" the help file.

Each tree has a root. Each NICE help file has a root section, which is located at the beginning of the file. The root has a name, which serves as a file label. The root name of a NICE Processor help file is usually the Processor name.

Logically subordinate to the root are the *topic keys*, which identify primary subjects such as command names. Topic keys may in turn have *subtopic keys* which identify secondary subjects such as command components. Subtopic keys may in turn have subordinate "subsubtopic" keys, and so on. However, all help files written so far for NICE Processors have not gone beyond the subtopic level.

As an example, an extract of a help file written for the DBEM2 Processor presented in Appendix C will be used. The example file contains the following three-level information tree:

```
DBEM2 (root)
  BUILD
  CLEAR
  DEFINE
    BOUNDARY_CONDITIONS
    ELEMENTS
    FIELD_LOCATIONS
    MATERIAL
    PRESTRESS
    SEGMENTS
    SYMMETRY_CONDITIONS
  BUILD
  GENERATE
  SOLVE
  PRINT
    BOUNDARY_CONDITIONS
    COEFFICIENTS
    ELEMENTS
    FIELD_RESULTS
    MATERIAL
    PRESTRESS
    RHS
    RESULTS
    SEGMENTS
    SOLUTION
    SYMMETRY
  STOP
```

DBEM2 is the root name, which is the same as the Processor name. There are seven topic keys: BUILD, CLEAR, DEFINE, GENERATE, PRINT, SOLVE and STOP. The topic names are the same as the action verbs of the DBEM2 commands listed in §B.5.

Topic key DEFINE has seven subtopics identified by keys BOUNDARY_CONDITIONS ... SYMMETRY_CONDITIONS. These subtopic identifiers correspond to the second keyword in DEFINE commands (see §B.7). Similarly, topic PRINT has eleven subtopics that correspond to the second keyword in PRINT commands (see §B.9). The other topic keys have no subtopics.

Assume that CLIP has been told the name of the Processor help file through an HFILE directive (Volume II). For example, the Processor may have submitted an HFILE directive as a message via CLPUT (§2.4). Printing of help file sections may now be requested through HELP directives.

Some directive examples:

```
*HELP
*HELP DEFINE
*HELP DEFINE MATERIAL
```

The first example directive requests only root information. The second one requests general information on command **DEFINE**. The third one requests specific information on command **DEFINE MATERIAL**.

§C.3 ORGANIZATIONAL DETAILS

The present section describes the organizational details of NICE help files. The **DBEM2** help file is used throughout as expository example.

The Root Section

Here is a sensible implementation of the root section of the DBEM2 help file:

```
C=DECK AAAROOT
> DBEM2
? B^uild C^lear D^efine G^enerate P^rint So^lve St^op
*
.
.   Help on the DBEM2 Processor commands can be obtained by typing
.
.           HELP   Topic   Subtopic  ...
.
.   where Topics are command names and Subtopics variations of
.   particular commands.
!
!   Available Topics:
!
!   BUILD      CLEAR      DEFINE     GENERATE   PRINT      SOLVE
!   STOP
!
*
```

Now for the details.

First of all, a help file is normally prepared using a text editor. It must therefore be formatted and sequential. The logical structure of the file is controlled by *sentinel characters* that appear in the *first column* of each line. The help-file reader recognizes the following sentinel characters:

Symbol	*Name*
>	right angle bracket
<	left angle bracket
*	asterisk
?	question mark
.	period
!	exclamation mark

Any line that doesn't have one of the above sentinels is *ignored* by the help file reader. This fact has practical applications in the maintenance of help files with the help of the MAX preprocessors, since *master source code* statements that begin with C= are ignored by the file reader and therefore harmless.

Five of the sentinels have been used in the above example: >, ?, *, . and !. Now's the time to describe the functions they perform.

The first line, which has a left-angle-bracket sentinel, contains the name of the root: DBEM2. This name effectively *labels* the help file. As noted previously, for a Processor help file the label is usually the Processor name. The name string may be written free-field after the sentinel. Whenever any section of a help file is listed, this label is written out as a reminder of which help file is being displayed.

Next comes the question-mark lines. These list topic keys subordinate to the root. These keys may be written in *root* plus *extension* form with the two components being separated by a caret sign, as explained in §5.1. For example, DEFINE is a topic key with root D, so it can appear in the question-mark lines as

```
D^efine
```

All lowercase letters are automatically converted to upper case for comparison tests, so any mixture of uppercase of lowercase letters is acceptable. You can render the above as D^EFINE or d^efine or DEFine; it doesn't matter.

In question-mark lines topic keys are written free field, with blank separators. No particular arrangement is expected: topic keys may appear *in any order*; however, alphabetic ordering (as in the example above) is recommended for disciplined file maintenance. There is no limit on the number of question-mark lines.

The empty line with only an asterisk sentinel is called a *terminator*. It simply tells the help file reader that no more topic-key lines follow. (An explicit terminator is needed because of the sequential nature of the file and the no-backspacing constraint.)

The next group of lines are identified by period and exclamation-mark sentinels. This is the *root help text*, and contains the lines that will actually be printed if a one-word HELP directive is received. The text that appears on the users' terminal should be

```
Help on the DBEM2 Processor commands can be obtained by typing

        HELP   Topic   Subtopic  ...

where Topics are command names and Subtopics variations of
particular commands.

Available Topics:

BUILD      CLEAR      DEFINE     GENERATE   PRINT      SOLVE
STOP
```

The name enclosed between < and > is the file label. It will appear on *all* help displays.

The distinction between period and exclamation mark sentinels only becomes relevant if a help request names an unknown topic. For example, suppose that CLIP receives the directive

```
*HELP ZZZ
```

The response on your terminal will be

```
Sorry, no documentation on      ZZZ

Available Topics:

BUILD      CLEAR      DEFINE     GENERATE   PRINT      SOLVE
STOP
```

The "sorry" message comes from the help file reader. Following the message, the lines with exclamation-mark sentinels are listed. But how does the reader know that topic ZZZ doesn't exist without ever going beyond the root section? Because ZZZ was not listed in the topic-key dictionary (the question-mark sentinel lines, remember?).

Finally, the last blank line with only an asterisk sentinel is again a terminator; this now explicitly marks the end of the listable text section.

A Topic Section with Subordinates

Now suppose that the help request is

```
*HELP DEFINE
```

The file reader begins searching the topic-key dictionary in the root. Satisfied that the topic exists, it speeds past the root to plunge deeper into the help file with a single-minded objective: to find the DEFINE section. This is how such a section may look:

```
C=DECK DEFINE
> D^EFINE
? B^oundary_conditions F^ield_locations M^aterial P^restress
? Se^gments Sy^mmetry_conditions
*
.   DEFINE
.
.   The DEFINE command introduces an input data section.
.
.   Format:
.
.                       DEFINE  What
.
.   where keyword   What  identifies the input data section that
.   follows.  The section consists of subordinate commands terminated
```

```
.   by an END command.   The legal keywords are listed below as
.   subtopics.
!
!   Available Subtopics:
!
!      BOUNDARY_CONDITIONS     ELEMENTS          FIELD_LOCATIONS
!      MATERIAL                SEGMENTS          PRESTRESS
!      SYMMETRY_CONDITIONS
!
*
```

The section replicates the basic structure of the root section in many respects. The first line has the by now familiar right-angle-bracket sentinel with the topic identifier written in "root plus extension" form. When the help file reader detects this combination it begins paying attention to the material that follows.

Next are several lines with question-mark sentinels. These list all ***subtopic keys*** subordinate to DEFINE, so they form a subtopic dictionary. This subsection is terminated by an asterisk-sentinel line. Then comes the listable information: lines with period and exclamation mark sentinels, the whole being closed by another terminator line. So you can pretty much guess that in response to the *HELP DEFINE directive, here is what you will see:

```
DEFINE

The DEFINE command introduces an input data section.

Format:

                    DEFINE  What

where keyword   What  identifies the input data section that
follows.  The section consists of subordinate commands terminated
by an END command.   The legal keywords are listed below as
subtopics.

Available Subtopics:

   BOUNDARY_CONDITIONS     ELEMENTS          FIELD_LOCATIONS
   MATERIAL                SEGMENTS          PRESTRESS
   SYMMETRY_CONDITIONS
```

A Topic Section With No Subordinates

Topic CLEAR of the example help file has no subordinate keys (*i.e*, no subtopics). The help file section has therefore a simpler structure:

```
      subroutine CLEAR
      call  CLPUTW ('ON')
      return
      end
```

The question-mark sentinel line is empty, which indicates no subtopics, and the exclamation-mark lines are also missing. But there is a new important thing at the end of the section: a line that has only a left-angle-bracket sentinel. This is necessary for *ordered traversal* of the information tree: it means that there are no more subordinate topics and we must "back up". (In computer science terminology: we have reached a *leaf node.*) We shall go over this important topic in detail in §C.4.

A Subtopic Section

If you have followed the explanation so far, you should have no trouble with this one. Let's assume the directive is

```
*HELP DEFINE MATERIAL
```

Here is the corresponding section:

```
C=DECK DEFINEMAT
> M^aterial
?
*
.   DEFINE
.      MATERIAL
.
.      The DEFINE MATERIAL command introduces subordinate material
.      property commands of the form
.
.                        EM =  em
.                        PR =  nu
.
.      where  em  is the elastic modulus and  nu  is Poisson's
.      ratio.  Terminate these commands with an END command.
.
*
<
```

This is quite similar to the CLEAR topic because it has no subordinates. Note again the left-angle-bracket termination line.

§C.4 PUTTING IT ALL TOGETHER

To assemble the complete help file one has to pay attention to some nesting concepts. This can be more easily understood by looking at the information tree for the example help file rendered as a hierarchical diagram:

```
root
  BUILD
  CLEAR
  DEFINE
    BOUNDARY_CONDITIONS
    ELEMENTS
    MATERIAL
    FIELD_LOCATIONS
    PRESTRESS
    SEGMENTS
    SYMMETRY
  GENERATE
  PRINT
    BOUNDARY_CONDITIONS
    COEFFICIENTS
    ELEMENTS
    FIELD_RESULTS
    MATERIAL
    PRESTRESS
    RESULTS
    RHS
    SEGMENTS
    SOLUTION
    SYMMETRY
  SOLVE
  STOP
```

This hierarchical tree actually defines the order in which the sections are juxtaposed to form the help file. First comes the root section, then a topic section, then its subordinate subtopics, then another topic section, and so on.

There is in fact considerable more latitude than the above structure suggests. Things at the same level need not be alphabetically ordered: they may actually appear in any order. For example, the BUILD section doesn't have to be the section that follows the root; you can make CLEAR or DEFINE or STOP the first one. Similarly, there is no need for the subordinate subtopics of, say, DEFINE to be alphabetically ordered. However, you cannot put the DEFINE subtopics after PRINT and vice versa: adjacency implies dependency.

The alphabetical ordering used in the example above is nonetheless recommended for maintaining help files, as it simplifies the work involved in inserting new help sections

corresponding to commands or command options you have just added to the Processor. Particularly important is the maintenance of *help file consistency* between dictionaries and topic sections. For example, suppose you have added a PLOT command to DBEM2. After writing a PLOT help section, you should not forget to go to the key dictionary located in the root section and add the PLOT keyword there.

Traversing the Help Tree

Whoever prepares a help file must be aware of the interplay between right-angle-bracket and left-angle-bracket sentinels. This understanding is necessary to fix the "lost help" difficulty discussed later in this subsection.

Much of the work of the help file reader is spent traversing across hundreds or even thousands of lines looking for the right key combinations. Conventionally the root is at *level zero*, topics at *level one*, subtopics at *level two*, and so on.

Upon leaving the root, traversal proceeds as follows: a right-angle-bracket sentinel increments the tree level by one unit; a left-angle-bracket sentinel decrements the tree level by one unit. Keeping track of the level is crucial for matching the right subject. For example, nothing prohibits the same subtopic key from appearing more than once in association with different subtopics. (In fact this is quite common, see *e.g.* DEFINE SEGMENT and PRINT SEGMENT in our sample file.)

To further illustrate the angle-bracket business, let us reproduce the example hierarchical diagram but now with > and < inserted in the proper places:

```
> root
  > BUILD <
  > CLEAR <
  > DEFINE
    > BOUNDARY_CONDITIONS <
    > ELEMENTS <
    > FIELD_LOCATIONS <
    > MATERIAL <
    > PRESTRESS <
    > SEGMENTS <
    > SYMMETRY < <
  > GENERATE <
  > PRINT
    > BOUNDARY_CONDITIONS <
    > COEFFICIENTS <
    > ELEMENTS <
    > FIELD_RESULTS <
    > MATERIAL <
    > PRESTRESS <
    > RESULTS <
    > RHS <
    > SEGMENTS <
    > SOLUTION <
    > SYMMETRY < <
  > SOLVE <
  > STOP < <
```

This diagram ought to make everything said so far perfectly clear.

A common problem encountered with new or updated help files is the "where is it?" syndrome. The user types, *e.g.* *HELP PRINT; no diagnostics appear but nothing comes out! This is usually caused by either leaving out a left-angle-bracket sentinel line, or by having one too many.

To pinpoint the trouble spot, try displaying topics stored nearer and nearer the root until the display appears. Then backtrack further from the root until the display suddenly disappears. This "bisection" troubleshooting technique is foolproof.

§C.5 IMPLEMENTING HELP COMMANDS AS MESSAGES

The user of a NICE Processor for which a help file exists can always get online help through the HFILE and HELP CLIP directives. But this is too much to expect from inexperienced users, who are hardly likely to know what a directive or a help file is, let alone what's the name of the latter. And such users are the ones in more need of help ...

What the beginner user of a Processor such as DBEM2 wants to do is to type HELP — not *HELP — and the root section appears magically on the screen; and to type HELP DEFINE and the DEFINE section appears on the screen. This can be easily implemented through messages as illustrated here for the DBEM2 Processor.

We first expand the original DOCOMMAND subroutine to install a HELP command (which may be abbreviated to H):

```
*
*       Top level command interpreter for DBEM2
*
      subroutine    DO_COMMAND  (verb)
*
      implicit      none
      character     key*8, verb*(*)
      logical       CMATCH
*
      key =   verb
      if (CMATCH (key, 'B^UILD'))         then
        call  BUILD
      else if (CMATCH (key, 'C^LEAR'))    then
        call  CLEAR
      else if (CMATCH (key, 'D^EFINE'))   then
        call  DEFINE
      else if (CMATCH (key, 'G^ENERATE')) then
        call  GENERATE
      else if (CMATCH (key, 'H^ELP'))     then
        call  HELP
      else if (CMATCH (key, 'P^RINT'))    then
        call  PRINT
      else if (CMATCH (key, 'SO^LVE'))    then
        call  SOLVE
      else if (CMATCH (key, 'ST^OP'))     then
        call  STOP
      else
        print *, '*** Illegal or ambiguous verb: ', key
      end if
      return
      end
```

Then we write subroutine HELP, which sends the necessary one-liners via CLPUT:

```
*
*     Scream for help
*
      subroutine    HELP
*
      implicit      none
      character*8   CCLVAL
      logical       first_entry
      save          first_entry
      data          first_entry /.true./
*
      if (first_entry)    then
        call  CLPUT ('*hf drd1:[felippa.manuals.clip.3.bem]dbem2.hlp')
        first_entry =  .false.
      end if
      call  CLPUT ('*help '//CCLVAL(2)//' '//CCLVAL(3))
      return
      end
```

On first entry to HELP, logical flag `first_entry` is true, and HELP informs CLIP of the help file name by sending an HFILE directive. (The full name of the help file has been assumed to be `drd1:[felippa.manuals.clip.3.bem]dbem.hlp` on a VAX/VMS system.)

Then the subroutine manufactures a HELP directive by catenating keywords entered by the user and submits it as a message. A maximum of two keywords after HELP has been assumed; it should be fairly obvious how to extend the construction to as many help levels as necessary.

§C.6 THE EXAMPLE HELP FILE

We conclude the Appendix with a listing of the complete DBEM2 help file. The reader should pay no attention to the C=deck statements; these are used only for file maintenance and are ignored by the CLIP help file reader.

```
C=DECK AAAROOT
> DBEM2
? B^uild C^lear D^efine G^enerate P^rint So^lve St^op
*
.
.   Help on the DBEM2 Processor commands can be obtained by typing
.
.           HELP   Topic   Subtopic  ...
.
.   where Topics are command names and Subtopics variations of
.   particular commands.
!
!   Available Topics:
!
!   BUILD      CLEAR      DEFINE     GENERATE   PRINT      SOLVE
!   STOP
!
*
C=DECK BUILD
> B^UILD
?
*
.   BUILD
.
.   This command builds the Boundary Element tables of the discrete
.   model.
.
.   Format:
.
.                       BUILD
.
.   The BUILD command must be given after the problem-definition
.   input data is complete, and before the GENERATE command.
.
*
<
C=DECK CLEAR
> C^LEAR
?
*
.   CLEAR
.
.   This command clears all problem definition arrays and sets
.   certain defaults.
.
.   Format:
```

```
.
.                        CLEAR
.
.    The CLEAR command is particularly useful if you are solving
.    several unrelated problems in one run.
.
*
<
C=DECK DEFINE
> D^EFINE
? B^oundary_conditions F^ield_locations M^aterial P^restress
? Se^gments Sy^mmetry_conditions
*
.    DEFINE
.
.    The DEFINE command introduces an input data section.
.
.    Format:
.
.                        DEFINE  What
.
.    where keyword   What  identifies the input data section that
.    follows.  The section consists of subordinate commands terminated
.    by an END command.   The legal keywords are listed below as
.    subtopics.
!
!    Available Subtopics:
!
!       BOUNDARY_CONDITIONS     ELEMENTS          FIELD_LOCATIONS
!       MATERIAL                SEGMENTS          PRESTRESS
!       SYMMETRY_CONDITIONS
!
*
C=DECK DEFINEBOU
> B^oundary_conditions
?
*
.    DEFINE
.       BOUNDARY_CONDITIONS
.
.       The DEFINE BOUNDARY_CONDITIONS commands introduces
.       BC commands that specify displacements and/or stresses on
.       boundary segments.  The BC commands have the form
.
.        SEG=iseg {SS = sig_s | SD=u_s} {NS=sig_n | ND = u_d}
.
.       in which  iseg  is the segment number.  SS means shear stress,
.       SD shear displacement, NS normal stress and ND normal
.       displacement; the value that follows is the prescribed
.       value.  For stresses, the value is the resultant force.
.       Terminate these commands with an END command.
.
.       If no BC is specified for a segment, a stress-free condition
```

```
.      is assumed.
.
*
<
C=DECK DEFINEELE
> E^lements
?
*
.   DEFINE
.      ELEMENTS
.
.      The DEFINE ELEMENTS command introduces subordinate commands
.      that specify into how many boundary elements segments are to
.      be subdivided.  These commands have the form:
.
.         SEG = iseg1, ... isegk   ELEM = ne1, ... nek
.
.      This specifies that segment  iseg1  is to be subdivided into
.      ne1 (ge 1) boundary elements,  segment  iseg2  into
.      ne2  elements, and so on.   Terminate these commands with
.      and END command.
.
*
<
C=DECK DEFINEFIE
> Field_locations
?
*
.   DEFINE
.      FIELD_LOCATIONS
.
.      The DEFINE FIELD_LOCATIONS command introduces subordinate
.      commands that specify field lines at which stresses and
.      displacements are to be evaluated later in response to a
.      PRINT RESULTS/FIELD  command.  These commands have the form
.
.         LINE=ilin  FIRST=xf,yf  LAST=xl,yl  [POINTS=nintpts]
.
.      LINE is a field-line identification  number (1 to 100).
.      The line extends from (xf,yf) to (xl,yl).  The optional POINTS
.      phrase specifies that output is required at  nintpts
.      intermediate points to be inserted, equally spaced,
.      between the first and last point.  If the phrase is omitted,
.      no intermediate points are inserted.  Terminate these
.      commands with an END command.
.
*
<
C=DECK DEFINEMAT
> M^aterial
?
*
.   DEFINE
```

```
.      MATERIAL
.
.      The DEFINE MATERIAL command introduces subordinate material
.      property commands of the form
.
.                     EM =  em
.                     PR =  nu
.
.      where  em  is the elastic modulus and  nu  is Poisson's
.      ratio.  Terminate these commands with an END command.
.
*
<
C=DECK DEFINEPRE
> P^restress
?
*
.   DEFINE
.      PRESTRESS
.
.      The DEFINE PRESTRESS command introduces subordinate
.      commands that specify a uniform initial stress
.      (prestress) state.  These commands have the form
.
.                      SXXO = sig_xx
.                      SYYO = sig_yy
.                      SXYO = sig_xy
.
.      The SXXO command specifies the sigma_xx prestress component,
.      and so on.  Nonzero prestress data is particularly useful in
.      unbounded-domain problems, for which it takes the role of
.      conditions at infinity.  Terminate this data with an END command.
.
.      If no prestress data is specified, the initial state
.      is assumed to be stress free.
.
*
<
C=DECK DEFINESEG
> S^egments
?
*
.   DEFINE
.      SEGMENTS
.
.      The DEFINE SEGMENTS command introduces subordinate commands
.      that specify the geometry of boundary segments on which boundary
.      elements will be located.  These commands have the form
.
.          SEG=iseg   BEGIN=xb,yb  END=xe,ye
.
.      in which  iseg  is the segment identification number
.      (1 to 100).  The segment extends from (xb,yb) to (xe,ye)
```

```
.       Terminate these commands with an END command.
.
.       The BEGIN/END specification establishes a boundary traversal
.       sense.  The traversal should be clockwise if you are solving
.       a finite-domain problem enclosed by the segmented boundary;
.       counterclockwise if you are solving an unbounded domain
.       problem (e.g a cavity) outside the segmented boundary.
.
*
<
C=DECK DEFINESYM
> Sy^mmetry_conditions
?
*
.   DEFINE
.      SYMMETRY_CONDITIONS
.
.      The DEFINE SYMMETRY_CONDITIONS command introduces subordinate
.      commands that specify symmetry conditions about 1 or 2 axes
.      parallel to the coordinate axes.   These commands have the
.      form
.
.                        XSYM = a
.                        YSYM = b
.
.      The XSYM command specifies  x=a  as an axis of symmetry,
.      and the YSYM command specifies  y=b  as an axis of
.      symmetry.   Terminate this information with an END command.
.
.      If no symmetry conditions are specified, no symmetry
.      conditions are assumed to hold.
.
*
<
<
C=DECK GENERATE
> G^enerate
?
*
.   GENERATE
.
.   This command causes the discrete element equations, which consist
.   of the influence coefficient matrix and the right-hand-side
.   (forcing) vector, to be generated.
.
.   Format:
.
.                        GENERATE
.
.   The GENERATE command must be given after a BUILD command but
.   before a SOLVE command.
.
*
```

```
<
C=DECK PRINT
> P^rint
? B^oundary_conditions  C^oefficients  F^ield_locations
? M^aterial  P^restress
? R^hs  Re^sults
? Se^gments  S^olution  Sy^mmetry_conditions
*
.   PRINT
.
.   The PRINT command requests printing of input data,
.   model definition data, or results data.
.
.   Format:
.
.                       PRINT  What
.
.   where keyword   What  identifies what is to be printed.
.   The legal keywords are listed below as subtopics.
!
!   Available Subtopics:
!
!      BOUNDARY_CONDITIONS    COEFFICIENTS           ELEMENTS
!      FIELD_LOCATIONS        MATERIAL               PRESTRESS
!      SEGMENTS               RESULTS                RHS
!      SOLUTION               SYMMETRY_CONDITIONS
!
*
C=DECK PRINTBOU
> B^oundary_conditions
?
*
.   PRINT
.      BOUNDARY_CONDITIONS
.
.      The PRINT BOUNDARY_CONDITIONS command prints the stress/
.      displacement boundary conditions in effect for all defined
.      boundary elements.
.
*
<
C=DECK PRINTCOE
> C^oefficients
?
*
.   PRINT
.      COEFFICIENTS
.
.      The PRINT COEFFICIENT command prints the matrix of
.      boundary influence coefficients produced by a GENERATE
.      command.  Primarily used for debugging.
.
*
```

```
<
C=DECK PRINTELE
> E^lements
?
*
.   PRINT
.      ELEMENTS
.
.      The PRINT ELEMENTS command prints the Boundary Element data
.      produced by the last BUILD command.
.
*
<
C=DECK PRINTFIE
> F^ield_locations
?
*
.   PRINT
.      FIELD_LOCATIONS
.
.      The PRINT FIELD_LOCATIONS command prints information on
.      field lines at which stresses and displacements are to be
.      computed in response to a PRINT RESULTS /F  command.
*
<
C=DECK PRINTMAT
> M^aterial
?
*
.   PRINT
.      MATERIAL
.
.      The PRINT MATERIAL command prints material property data.
.
*
<
C=DECK PRINTPRE
> P^restress
?
*
.   PRINT
.      PRESTRESS
.
.      The PRINT PRESTRESS command prints initial stress data.
.
*
<
C=DECK PRINTRES
> R^esults
?
*
.   PRINT
.      RESULTS
```

```
.
.       An unqualified PRINT RESULTS command prints computed stresses
.       and displacements at boundary element midpoints.
.
.       If the command is qualified with keyword FIELD, stresses
.       and displacements at specified field locations will be
.       computed and printed.
.
*
<
C=DECK PRINTRHS
> R^hs
?
*
.    PRINT
.       RHS
.
.       The PRINT RHS command prints the right-hand-side (boundary
.       force) vector produced by a GENERATE command.  Primarily used
.       in debugging situations.
.
*
<
C=DECK PRINTSEG
> Se^gments
?
*
.    PRINT
.       SEGMENTS
.
.       The PRINT SEGMENTS command prints geometric information on
.       defined boundary elements as well as the number of
.       boundary elements per segment.
.
*
<
C=DECK PRINTSOL
> So^lution
?
*
.    PRINT
.       SOLUTION
.
.       The PRINT SOLUTION command prints the boundary solution vector
.       produced by a SOLVE command.  Primarily used in debugging
.       situations.
.
*
<
C=DECK PRINTSYM
> Sy^mmetry_conditions
?
*
```

```
.   PRINT
.      SYMMETRY_CONDITIONS
.
.      The PRINT SYMMETRY_CONDITIONS command  prints symmetry
.      conditions data.
.
*
<
<
C=DECK SOLVE
> S^olve
?
*
.   SOLVE
.
.   This command causes the discrete element equations to be
.   solved for the boundary element unknows.
.
.   Format:
.
.                        SOLVE
.
.   The SOLVE command must be given after a GENERATE command.
.
*
<
C=DECK STOP
> St^op
?
*
.   STOP
.
.   This command terminates the execution of the Processor.
.
.   Format:
.
.                        STOP
.
*
<
<
```

THIS PAGE LEFT BLANK INTENTIONALLY.

D

Low-level Utilities

§D.1 GENERAL DESCRIPTION

This Appendix presents some low-level utilities which are not part of CLIP itself, but are heavily used by CLIP as well as by other components of the NICE architecture. Most of these utilities deal with character manipulation.

The calling sequences of these utilities is described here because they may be useful in programming the Processor shell. We have seen some examples in Appendix A.

§D.2 CONVERT CHARACTER TO HOLLERITH: CC2H

CC2H converts a FORTRAN 77 character string to a Hollerith string. The first destination character is assumed to be word aligned.

Calling Sequence

```
CALL CC2H (C, H, N)
```

Input Arguments

C	Source character string.
N	Number of characters to be moved. No operation if N $\leq$ 0.

Output Argument

H	Receiving Hollerith string (typed integer, floating-point or logical in the main program). Characters are stored in H beginning at the leftmost location. If H is of INTEGER or REAL type, this is necessarily word-aligned. CC2H *does not* blankfill H, however.

REMARK D.1

The implementation of CC2H has turned out to be surprisingly machine-dependent. So far, five implementations have had to be written for five computers (CDC, CRAY, IBM, UNIVAC and VAX). Three versions are presented below, as the techniques followed may be useful for similar circumstances.

REMARK D.2

The VAX implementation, which takes advantage of the LOGICAL*1 data type provided by the VAX-11 FORTRAN compiler, is the simplest and most efficient:

```
      subroutine    CC2H
     $              (c, h, n)
      implicit      none
      logical*1     h(*)
      character     c(*)
      do 3000  k = 1,n
        h(k) =   ichar(c(k))
3000   continue
      return
      end
```

REMARK D.3

The implementation for a word-addressable machine must make use of bit-manipulation (Boolean) functions provided by the host FORTRAN compiler. This is illustrated by the CRAY version:

```
      subroutine    CC2H
     $              (c, h, n)
      character*1   c(*)
      integer       h(*), ich, jch, k, n, iwd, lcs
      do 3000  k = 1,n
        iwd =     (k-1)/8 + 1
        lcs =     8*( 8*iwd - (k-1))
        ich =      shift(ichar(c(k)),56)
        jch =      and (h(iwd),shift(mask(72),lcs) )
        h(iwd) =  or (jch, shift(ich,lcs))
 3000   continue
      return
      end
```

REMARK D.4

On byte addressable machines in which the compiler provides the **LOGICAL*1** data type but prohibits storing an integer into it, one may use an internal READ construction illustrated by the IBM version:

```
      subroutine    CC2H
     $              (c, h, n)
      character*(*) c
      logical*1     h(*)
      m1 =    0
      do 3000  k = 1,(n+127)/128
        m2 =     min(m1+n-128*(k-1),m1+128)
        read (c(m1+1:m2),'(128A1)'),  (h(i),i=m1+1,m2)
        m1 =     m2
 3000   continue
      return
      end
```

REMARK D.5

The above versions have been extracted from the master source code of **CC2H** via the preprocessor **MAX**.

EXAMPLE D.1

Four characters per word are assumed.

```
INTEGER H(5)
...
CALL CC2H ('X-label', H, 7)
```

Word H(1) receives 'X-la' while characters 1–3 of H(2) receive 'bel'. The remaining character positions in H are not altered.

§D.3 CONVERT CHARACTER TO OFFSET HOLLERITH: CC2HO

CC2HO converts a FORTRAN 77 character string to a Hollerith string. The first destination character need not be word aligned.

Calling Sequence

```
CALL CC2HO (C, H, J, N)
```

Input Arguments

C	Source character string.
J	Offset of first receiving character in H. May be zero through NCWORD-1, where NCWORD is the number of characters per word. If J is outside this range the results are unpredictable.
N	Number of characters to be moved. No operation if N $\leq$ 0.

Output Argument

H	Receiving Hollerith string (typed integer, floating-point or logical in the main program). Characters are stored in H beginning at J characters from the leftmost (word aligned) character position.

REMARK D.6

CALL CC2HO (C, H, 0, N) (zero third argument) is the same as CALL CC2H (C, H, N).

EXAMPLE D.2

Four characters per word are assumed.

```
LOGICAL QQ(3)
...
CALL CC2H ('Level:     ', QQ, 12)
CALL CC2HO ('22', QQ(2), 3, 2)
```

On return from CC2HO, QQ will contain 12HLevel: 22 .

§D.4 CONVERT HOLLERITH TO CHARACTERS: CH2C

CH2C converts a Hollerith string to a FORTRAN 77 character string. The first source character is assumed to be word aligned.

Calling Sequence

```
CALL CH2C (H, C, N)
```

Input Arguments

H	Source Hollerith string. Array H may be of type integer, floating-point or logical in the calling program.
N	Number of characters to be moved. No operation if N $\leq$ 0.

Output Argument

C	Receiving character string.

REMARK D.7

The implementation of CH2C is similar to that of CC2H (§D.2), and is likewise machine-dependent. So far, five implementations have had to be written for five computers (CDC, CRAY, IBM, UNIVAC and VAX).

EXAMPLE D.3

Four characters per word are assumed.

```
INTEGER  H(3)
CHARACTER*12 CH
DATA     H /4Habcd, 4Hefgh, 4Hijkl/
...
CH = '------------'
CALL CH2C (H(2), C(3:7), 4)
```

On return from CH2C, CH will contain '--efgh------'.

§D.5 CONVERT OFFSET HOLLERITH TO CHARACTER: CHO2C

CHO2C converts a Hollerith string to a FORTRAN 77 character string. The first source character need not be word aligned.

Calling Sequence

```
CALL CHO2C (H, J, C, N)
```

Inputs Arguments

H	Source Hollerith string.
J	Offset of first source character in H. May be zero through NCWORD-1, where NCWORD is the number of characters per word. If J is outside this range the results are unpredictable.
N	Number of characters to be moved. No operation if N $\leq$ 0.

Output Argument

C	Receiving character string.

REMARK D.8

CALL CHO2C (H, 0, C, N) (zero second argument) is the same as CALL CH2C (H, C, N).

EXAMPLE D.4

Four characters per word are assumed.

```
INTEGER  H(3)
CHARACTER*12 CH
DATA     H /4Habcd, 4Hefgh, 4Hijkl/
...
CH = '------------'
CALL CHO2C (H(2), 2, C(3:7), 4)
```

On return from CHO2H, CH will contain '--ghij------'.

§D.6 COMPARE KEYWORDS: CMATCH

CMATCH is a logical function that compares two character strings: an alleged keyword and an internal keyword, for equality. The internal keyword consist of a "root" and an optional "extension" portion. These two portions may be separated by a caret character as described in §5.1. CMATCH first compares root characters, and reports failure if no match occurs. If root match is achieved, it continues comparing extension characters until:

(a) A mismatch is found;

(b) The input or output key is exhausted; or

(c) A blank character is found in either key.

The alleged key may also contain wild characters: a percent sign matches any character at that particular position in the internal key, and a trailing asterisk matches all characters that follow.

Typical Calling Sequence

```
LOGICAL CMATCH
...
IF (CMATCH (KEY1, KEY2)) THEN
...
```

Input Arguments

KEY1	Alleged keyword; typically this is entered by the user.
KEY2	Internal keyword against which KEY1 is compared.

Function Return

CMATCH	.TRUE. if equality verified; else .FALSE.

REMARK D.9

This is the present implementation of CMATCH as extracted from the master source code via MAX:

```
      logical function  CMATCH
     $                  (key1, key2)
      implicit          none
      character*(*) key1, key2
      character         ch1, ch2
      integer           i, j, root
      i =           0
      root =        1
      CMATCH =      .TRUE.
      do 2000    j = 1,len(key2)
        ch2 =         key2(j:j)
        if (ch2 .eq. '^')                   then
          root =    0
          go to 2000
        end if
        if (ichar(ch2) .ge. ichar('a') .and.
     $      ichar(ch2) .le. ichar('z'))          then
          ch2 =     char(ichar(ch2)-(ichar('a')-ichar('A')))
          root =    0
        end if
        ch1 =       ' '
        i =         i + 1
        if (i .le. len(key1))               ch1 = key1(i:i)
        if (ch1 .eq. ' ' .and. root .eq. 0)  return
        if (ch1 .eq. '*')                   return
        if (ch1 .eq. '%')                   ch1 = ch2
        if (ch1 .ne. ch2)                   then
          CMATCH =      .FALSE.
          return
        end if
        if (ch2 .eq. ' ')                 return
 2000   continue
      return
      end
```

There are no machine dependencies except for the IMPLICIT NONE statement, which is provided only by the best FORTRAN compilers.

EXAMPLE D.5

```
CMATCH ('COPY', 'COPY')              returns     .TRUE.
CMATCH ('CORRECT', 'CO^PY')          returns     .FALSE.
CMATCH ('COPYALL', 'CO^PY')          returns     .TRUE.
CMATCH ('CO*', 'COM^PARE')           returns     .TRUE.
CMATCH ('CO%%PARE', 'COM^PARE')      returns     .TRUE.
```

§D.7 FIND BATCH OR INTERACTIVE: FBI

FBI finds whether the run is batch or interactive.

Calling Sequence

```
CALL FBI (RUNMOD)
```

Input Arguments

None.

Output Argument

RUNMOD — An integer variable that receives the run mode indicator.

0: batch mode.

>0: interactive mode.

On the VAX there is further breakdown of the interactive case:

2: interactive mode, input coming from terminal. (also called conversational mode).

1: interactive mode, input from source other than terminal.

REMARK D.10

As can be expected, the implementation of FBI is extremely machine dependent, because run type information has to be provided by the operating system. Here is the VAX/VMS version, which is currently the most elaborate one:

```
      subroutine    FBI
     $              (runmod)
      implicit      none
      integer       runmod
      integer*2     itmlst(8)
      integer*2     item_code, buffer_length
      integer       buffer_addr, sts_flags, TERM_TEST
      character*12 logical_name
      equivalence  (buffer_addr,itmlst(3))
      data          itmlst /4,'0305'X,6*0/
      runmod =   0
      buffer_addr = %loc(sts_flags)
      call   SYS$GETJPI (,,,itmlst,,,)
      if (iand(sts_flags,'4000'X) .eq. 0)  then
        runmod =  TERM_TEST ('SYS$INPUT') + 1
      end if
      return
      end
```

```
      integer function term_test (logical_name)
      implicit  none
      integer   DIB$B_DEVCLASS, DIB$K_LENGTH, DC$_TERM
      character*(*) logical_name
      parameter (DIB$B_DEVCLASS  = '00000004'X)
      parameter (DC$_TERM        = '00000042'X)
      byte      dib_record (0:4)
      byte dib_b_devclass
      character*5 chr_record
      equivalence (chr_record,dib_record)
      equivalence   (dib_b_devclass, dib_record(DIB$B_DEVCLASS))
      call      SYS$GETDEV(logical_name,,chr_record,,)
      term_test =  0
      if (dib_b_devclass .eq. DC$_TERM) term_test = 1
      return
      end
```

REMARK D.11

And here is the CDC/NOS implementation:

```
      subroutine    FBI
     $              (runmod)
      integer       runmod
      integer       runtyp
      runmod =   0
      call       JOBSTAT (6HJOBORG, runtyp)
      if (runtyp .eq. 3)               runmod = 1
      return
      end
```

REMARK D.12

Some operating systems refuse to give information of this type. In such a case **FBI** returns zero.

§D.8 GET LENGTH EXCLUDING TRAILING BLANKS: LENETB

Function LENETB receives a character string as an argument, and returns its length when all trailing blanks are excluded.

Calling Sequence

```
L = LENETB (TEXT)
```

Input Argument

TEXT

Character string.

Function Return

LENETB — The length of the argument string excluding all trailing blanks found when scanning it backwards starting at the passed length. If TEXT contains only blanks, a length of zero is returned.

REMARK D.13

Here is the current implementation of LENETB:

```
      integer function LENETB
     $                 (c)
      implicit         none
      character*(*) c
      integer          i
      LENETB =  len(c)
      do 2000  i = len(c),1,-1
        if (c(i:i) .ne. ' ')          return
        LENETB =  i-1
 2000   continue
      return
      end
```

EXAMPLE D.6

LENETB(' Hi there ') returns 9, but LENETB(' ') returns zero.

THIS PAGE LEFT BLANK INTENTIONALLY.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-178386		

4. Title and Subtitle	5. Report Date
The Computational Structural Mechanics Testbed Architecture Volume III - The Interface	December 1988
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Carlos A. Felippa	LMSC-D878511
9. Performing Organization Name and Address	10. Work Unit No.
Lockheed Missiles and Space Company, Inc. Research and Development Division 3251 Hanover Street Palo Alto, California 94304	505-63-01-10
	11. Contract or Grant No.
	NAS1-18444
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes

Current affiliation: Carlos A. Felippa, Center for Space Structures and Controls, Campus Box 429, University of Colorado, Boulder, CO 80309-0429

Langley Technical Monitor: W. Jefferson Stroud

16. Abstract

This is the third of a set of five volumes which describe the software architecture for the Computational Structural Mechanics Testbed. Derived from NICE, an integrated software system developed at Lockheed Palo Alto Research Laboratory, the architecture is composed of the command language (CLAMP), the command language interpreter (CLIP), and the data manager (GAL). Volumes I, II, and III (NASA CR's 178384, 178385, and 178386, respectively) describe CLAMP and CLIP and the CLIP-processor interface. Volumes IV and V (NASA CR's 178387 and 178388, respectively) describe GAL and its low-level I/O. CLAMP, an acronym for Command Language for Applied Mechanics Processors, is designed to control the flow of execution of processors written for NICE. Volume III describes the CLIP-Processor interface and related topics. It is intended only for processor developers.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
Structural analysis software Command language interface software Data management software	Unclassified—Unlimited Subject Category 39

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	212	A10

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171